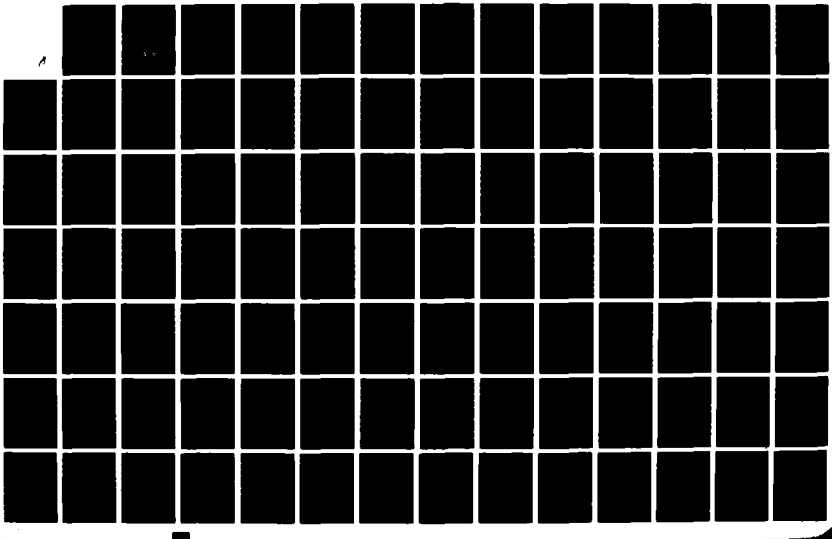
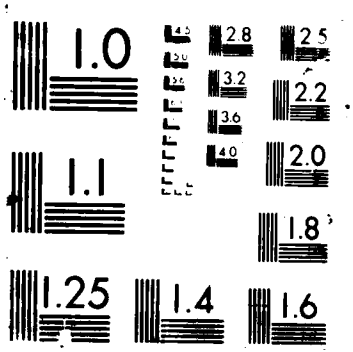
AD-A187 405 UNIVERS PRODUCT PHASE 1(U) ONTOLOGIC INC BILLERICA MA 1/2
M J VILOT ET AL. 27 APR 87 N00014-86-C-0685

UNCLASSIFIED F/G 12/7 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

Contract N00014-86-C-0605

SBIR Phase I Final Report
UNIVERS Product

Michael J. Vilot
Robert M. Strong

Ontologic, Inc.
47 Manning Rd., Billerica MA 01821

ABSTRACT

This document records the results of Phase I of a three-phase product development effort. The purpose of this project is to investigate extensible databases for design applications. These applications have complex data modeling needs not adequately met by existing database systems. Our goal is to combine the *abstraction* and *modularity* of modern programming languages and the *persistent storage* management of databases with the *inheritance/refinement* mechanism of object-oriented systems to provide an extensible database product.

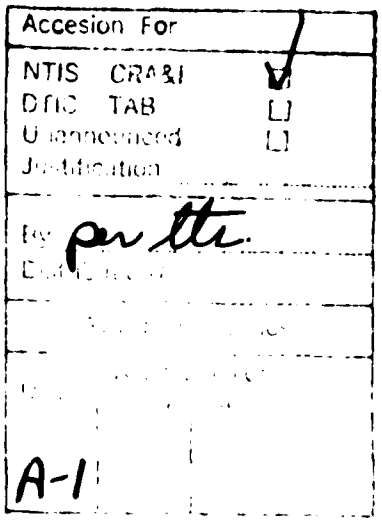
The objectives for this first phase are to investigate data modeling and representation requirements for extensibility. Our primary focus is on programming language access to the extensibility mechanisms of an object-oriented database. This Phase I project produced a definition of requirements for the product we call UNIVERS — the UNIfication of programming language and database technology, with the VERSatility of object-oriented systems. , Phase I also establishes a high-level (architectural) design for UNIVERS, including a description of the programming language/database interface, and an estimate of the feasibility of Phase II.

The resulting product is an Ada® language interface providing access to Ontologic's existing database product, Vbase. The Ontologic Vbase is an object-oriented database development platform targeted at the needs of the engineering design application builder, It is designed to serve as a foundation for MCAD, ECAD, and CIM applications. The existing product runs under 4.2 BSD UNIX™ on SUN 3™ workstations, and will soon be available on Digital Equipment's VMS™ operating system.

Potential UNIVERS applications include Government-sponsored ECAD design applications (for example, the VHSIC program) and the design needs of major Government contractors (for example, aerospace firms). We also think that Ada access to Vbase makes an excellent platform for CASE environments, including a sophisticated APSE for large software system development.

CONTENTS

LIST OF FIGURES



LIST OF TABLES

SBIR Phase I Final Report
UNIVERS Product

Michael J. Vilot
Robert M. Strong

Ontologic, Inc.
47 Manning Rd., Billerica MA 01821

1. Introduction

This project is Phase I of a 3-phase Defense Small Business Innovation Research (SBIR) program. It was undertaken in response to SBIR solicitation N86-7, entitled "Extensible Databases:"[DOD86]

> "Research and development in the database management systems area has traditionally concentrated on supporting business applications. Recently the design of database systems capable of supporting non-traditional application areas such as CAD/CAM and VLSI, scientific and statistical applications, expert database systems, and image/voice processing has emerged as an important direction of database systems research. These new applications differ from more conventional applications (e.g. transaction processing, record keeping) in a number of critical aspects, including data modeling requirements, processing functionality, concurrency control, recovery, access methods, and storage structures. A number of groups are attempting to meet the demands of these new applications by either building interfaces on top of existing relational database systems or by trying to extend the functionality of such systems. These approaches may appear promising, but are not likely to provide acceptable performance. Research is needed which will lead to **design methods** for database management systems that are *flexible* enough to permit *extensions* to operations such as data modeling, query processing, access methods, storage structures, concurrency control and recovery."

The work was performed for the Department of the Navy's Office of Naval Research, under Navy contract N00014-86-C-0605. This Final Report documents the results of Phase I of the project. It records the results of our efforts defining the project, and provides the background for explaining why we are addressing certain issues.

Intended Audience: Database researchers interested in extensibility. Programming language researchers interested in persistent storage. Design application builders with complex modeling requirements.

Document Objectives: To convey the results of the SBIR Phase I effort.

Knowledge Prerequisites: An understanding of Data Base Management Systems and programming languages. Familiarity with the Ada programming language.

Associated Documents:

Vbase Data Model Reference Manual.[Vba86a]
Vbase Language Reference Manuals.[Vba86b, Vba87a]
Vbase Functional Specification.[Vba86c]
Vbase Technical Overview.[Vba87b]

1.1 Project Overview

The purpose of this three-phase SBIR project is to research critical aspects of extensible databases. Figure 1-1 (see next page) summarizes the the relationship between activities, milestones, and documents for the UNIVERS project. Phase I investigates the technology and defines product requirements. It also evaluates the feasibility of the product. Phase II builds a working prototype of the product. Phase III commercializes the technology, making the product generally available and supporting it in the marketplace.

Ontologic has developed an effective solution to extensible databases, which we call the Vbase Object Manager. Our approach to the UNIVERS product is to design methods providing extensions to Vbase which will be useful to application developers. In particular, we address the issues of using the database in large design applications which have significant (but unique) data modeling and storage requirements.

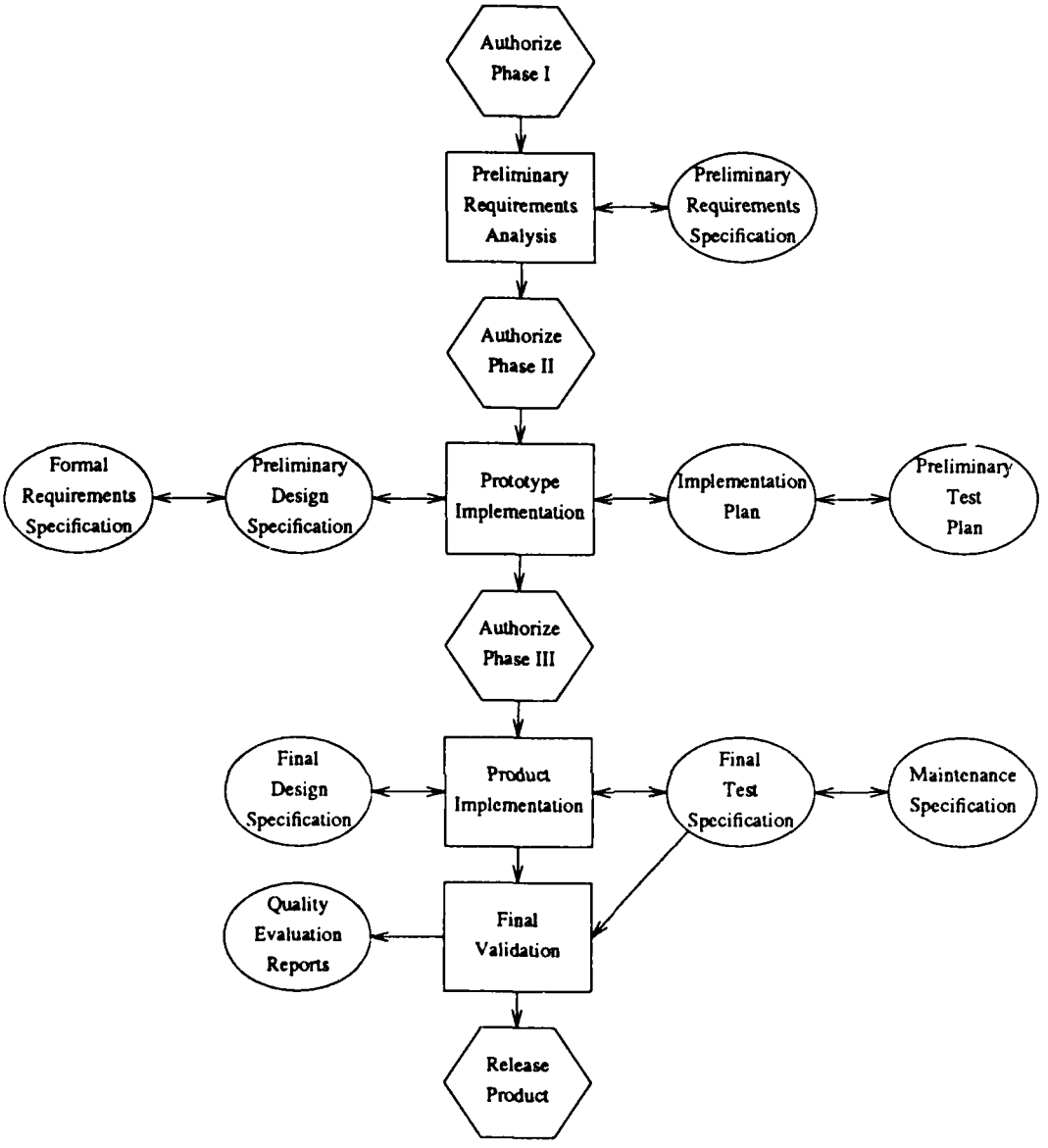


Figure 1-1. Development Process Model

This SBIR project focuses on the interface between our Vbase object-oriented data base product and the Ada programming language.[DOD83] Our goal is to provide an interface which makes effective use of this powerful object technology, yet which respects the philosophy of Ada and its compilation semantics. This approach meets the demanding needs of design applications with complex and evolving data modeling requirements.

1.1.1 Phase I Goals

When we applied for support under the SBIR Program, we were in the early stage of thinking about what an object-oriented database product should look like. Between the proposal and the time the Phase I Contract was awarded, the Vbase product development proceeded. By the time work started for the SBIR Phase I effort, we had designed and essentially developed what we had proposed in the original submission.

In an effort to salvage maximum value under the SBIR Contract awarded, and consistent with the basis of the original proposal, we have researched issues concerning the extension of our product with an additional language interface, and have selected Ada as the target. We have now completed the exploratory work of evaluating Ada

and methodologies for making our database product available to Ada programmers, and thus making an extensible database available to them.

We have found a unique level of support in Vbase for providing the Ada world with integrated, persistent storage of data. We found significant synergy between the Ada philosophy about abstraction and type-enforcement and same concepts in Vbase (which we've implemented as extensions to the C programming language[Ker78]). We have also begun to identify limitations intrinsic to Ada.

This effort, however, has converged on what appears to be a "best" strategy to providing object-oriented database support in a manner consistent with Ada philosophy. We approached the problem from the point of view of doing the best that can be done to support Ada programmers, yet accepts the limitations within which they normally work (such as static type checking).

1.1.2 Work Efforts

The Phase I effort investigates extensible database technology and defines requirements for UNIVERS. To this end, we pursued four activities:

- Examined design application needs to establish basic objectives and criteria for the system.
- Examined Ada language issues relevant to persistent object storage.
- Described an Ada/Object Oriented Data Base interface.
- Assessed the feasibility of building the interface.

This document is the only deliverable item for the first phase of the product development effort. It records the results of our requirements analysis and high-level (architectural) design.

1.1.3 Results

Phase I identified several issues for extensibility in databases. We feel that one of the central concerns is flexibility and extensibility in the data model. We selected an approach which combines the concepts of *abstraction* and *modularity* from programming languages; *persistent storage* from database systems; and *objects* from research prototypes.

Phase I also investigated the feasibility of building the Ada interface. The semantics of the two systems are similar in many respects, yet differ significantly in detail. We feel that the UNIVERS product is feasible, but that there are certain issues which require further study:

- Integration of the type systems of Ada and Vbase, and the degree of type checking available,
- The difficulty of translating Vbase Data Model information into Ada package structures in the Ada program library,
- The compatibility of specific Vbase constructs (for example, **iterators** and **exceptions**) with Ada,
- The interaction of representation specifications, storage **pragmas**, and the Ada compiler

The Phase II effort will build a prototype of the product, to examine these issues. We feel that Vbase, as an object-oriented database, embodies the necessary technology for database extensibility. The concepts of inheritance and refinement provide superior support for extensibility, with a database system providing performance which meets the most demanding needs of CAD support systems. We selected Ada as an interface to Vbase for two reasons:

1. It integrates, in one language, much of the support for abstraction and modularity we support in Vbase.
2. Its standardization guarantees wide availability and a great deal of consistency between implementations — it should be a reasonably stable interface.

The combination of Ada and Vbase provides a powerful system which meets the needs of design applications. The architecture we've defined presents the system in a flexible, yet consistent, package which should be easy to use for design application builders used to working with Ada. It also provides a unified Data Definition and Data Manipulation interface through a minimal extension of the Ada language.

1.2 Overall Structure of the Document

This report describes the extensible database issues, defines the integrated language, and presents the basic facilities of the UNIVERS product. It details the behavior of the system: the types, their syntax, and their semantics. It also includes examples of how the behaviors can support extensibility in the database.

The objective of the Phase I report is to check the interface description — whether it supports the necessary degree of extensibility, and whether it provides an adequate basis for implementation in Phase II. The primary focus of Phase II will be engineering an Ada interface to the existing product. The result will be a working prototype of an object-oriented language providing access to persistently stored objects.

1.2.1 Requirements

The first few sections of this report describe the requirements for the UNIVERS system:

Section 2	Background on design application needs and extensible database support issues.
Section 3	Basic concepts of abstraction, types and behavior, and object-oriented systems.
Section 4	Issues surrounding an Ada programming language interface to the Vbase system.

1.2.2 Architecture

The next few sections describe the alternatives and selected approach for the software to be developed in the Phase II effort:

Section 5	Approaches to an Ada/Vbase interface. A summary of our selected approach, which features a pre-processor and an interface package in the Ada program library.
Section 6	Pre-processor and the language constructs it recognizes.
Section 7	Ada/Vbase interface and the services it provides.

1.2.3 Extensibility Examples

The final sections demonstrate the concepts we've explored, by providing examples of database extensibility. The report concludes with a summary of our results.

Section 8	Extensibility examples, using the language, interface package, and data model we've defined.
Section 9	Summary of our analysis and conclusions of Phase I. Recommendations for Phase II.

1.2.4 Supporting Material

The other sections contain much of the detail from our Phase I work efforts:

Appendix A	Glossary of terms.
Appendix B	Definitions of acronyms.
Appendix C	References.
Appendix D	Grammar of the language which extends Ada for persistent object management.
Appendix E	Software listings for the recognizer developed to check the grammar and interface definition.
Appendix F	Kernel data model for an extensible database.

2. Background

This section discusses the background and rationale for the project. In this section, we examine the basic requirements for any proposed extensible database solution, and investigate the reasons traditional database and programming language systems fail to meet the needs of design applications. We considered the most demanding database support requirements: Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM) systems.

For the past several years, it has been recognized in the CAD/CAM community that currently available database management technology does not address basic problems of the engineering process:

- Complex relationships between design, manufacturing, and commercial databases.
- Designs themselves becoming an order of magnitude more complex with each decade.
- Explosive creation and mutation of user requirements and technology mandates.
- Different functional orientations on the same data, for example, pipes as structural parts and as hydraulic parts.
- Iterative design, requiring versions and alternatives.
- High-performance, high-resolution graphics display and manipulation.
- Heterogeneous hardware and software, collected in attempts to meet requirements without a single, integrated solution.
- Distributed computing, across offices and the country, internationally, and in space.

Lack of solutions to these problems have hindered growth and competitiveness of CAD/CAM engineering systems.

2.1 Design Support Needs

The strongest initial need for extensible databases comes from users of non-traditional data in the areas of CAD/CAM/CIM, and also VLSI design. These users have complex modeling requirements which require abstraction mechanisms not provided in traditional database management systems.

Engineering design is the process of building up a model of a complex artifact. The artifact itself may be decomposed into hundreds of other components. And at any level of the decomposition there may be several logical ways of looking at the design as well as several alternatives for its physical realization.

The database requirements of engineering design support applications are in some ways significantly different from those of general business applications. It must support a wide range of data types. Its concurrency control, and transaction management must support long transactions where deadlock and transaction back out are not acceptable. And it has to model the successive versions of evolving designs; their configurations and consistency.

2.1.1 Expressing Complex Models

The first requirement of a design support database management system is a data model which is strong enough to bring order to this complexity. In design applications, there are two components to this complexity: the large number of different data types, and variety of interrelationships (and attendant semantic constraints).

2.1.1.1 Diverse Data Types One key aspect of modern design systems is the ability to combine diverse types of data, most often text and graphics. An example of this is the research into the Notecards application by the CIA. This application allows reports containing text and graphics, most typically annotations of line drawings derived from satellite images, or annotations of maps. Studies by the agency have demonstrated that the ability to include visual representations along with the text of the report greatly decreases the chance that important information will be lost in the "filtering" process which takes place when analysts submit purely textual interpretations of visual information.

As another example, we have talked informally with researchers in medical imaging at the Dana Farber Cancer institute, who use multiple CAT scans to generate 3-D images of complex bone structures (such as hip socket joints and facial structures) for use in surgical simulation. These researchers expressed a strong desire for a system to

reduce the amount of custom software necessary to implement the simulations (currently it is all customized), and to provide greater efficiency of both the storage and run time areas in the system.

As a final example, tools for a Computer-Aided Software Engineering (CASE) environment need a full range of atomic and composite data types. One important type that appears in many CASE applications is directed graphs. CASE tools might store and manipulate parse trees, flow graphs, dependency graphs, and PERT charts. In today's DBMSs, one must store the graphs by storing the edges in relations, and must manipulate those graphs using standard relational operators. Many of the fast algorithms for manipulating directed graphs are not easily expressed using relational data structures and operations.

2.1.1.2 Complex Relationships Another area worth mentioning is semantic databases, sometimes referred to as knowledge bases. These areas are the foundation upon which expert systems are based and require systems which provide far greater support than traditional systems in many areas. Their greatest needs are in the incorporation of the operation abstraction and the support of multiple inheritance, which greatly enhances the semantic modeling capabilities of the system.

For example, a CAD application object can represent a machine part. The static properties of the part might be its geometry, material, weight and color. Its dynamic behavior might be specified by functions that calculate stress, deformation under load, thermal properties or anything else germane. The system should be able to represent the semantic structures **directly** in the database, typically derived from the raw data. For example, the "length" attribute could be calculated by procedure, rather than stored explicitly.

The DBMS should check the integrity of objects relative to their type definition. Database objects may have integrity constraints that are more complex than can be expressed in the type definition language. For example, one may have a style checking program that determines whether a document is consistent with an organization's standards.

The DBMS may offer a *trigger* mechanism (also called *demons* in AI systems), similar to exception handling mechanisms in programming languages. For example, one could define a trigger on document objects, which is activated by the "check-in" operation, and causes the style-checker to be invoked. Triggers can also be used as an alerting mechanism. For example, a user can check out an object for reading, and leave a trigger in case someone else wants to check out the same object for updating. The action part of the trigger simply sends a message to the first user.

2.1.2 Design Evolution

Commercial database management systems are what we term "shadow" systems. They track the real world. Since the real world has one current state, the database has one current state. That makes sense in the commercial applications these systems were designed to handle. In design support applications, however, it is precisely the versions and theoretical alternatives which are of interest.

Concurrency control in business DBMSs is based on the notion of **global consistency**. A *transaction* takes the database from one globally consistent state to another. In a design database, this notion of consistency is not useful. A design database may not achieve a globally consistent state for weeks or months; in fact, it may never do so over the period that it is useful in supporting the design process. The point at which it achieves consistency is by definition the point at which design is complete. At any particular point in its evolution, specific version of specific portions of the database may be consistent with some portions of the design, but not others.

2.1.3 Separation of Concerns

Two keys to flexible use of an extensible system revolve around the separation of concerns. The particular issues are Specification/Implementation concerns, and Specification/Representation concerns. Incremental additions (type characterizations, subtype inheritance) to the external behavior should be clearly separated from the implementation of the behavior. The implementation may include unique data structures and storage layouts, but this is not required. However, without a clear separation between behavioral specification and implementation, incremental additions to behavior would be much more difficult. Another element in extensibility is the distinction drawn between the specification of a type and its representation. This distinction allows *custom representations* to be defined for different types of data.

Without this separation, the application programmer would be forced to bind the representation details into the application, thus making it less portable and less maintainable (and also making it more difficult to "tune" the database for performance). There are representations which can be made highly efficient for each particular type of data, independent of the representations of the other data types:

- 2D and 3D graphics
- geometry
- solids models
- matrix-based analysis/simulation data
- cartographic data
- digitized images
- voice

In addition, an application may wish to use this feature to integrate data from a "foreign" DBMS or application. The representation of index and data layout provides an avenue for retaining compatibility with existing data. These data may represent a significant investment, particularly with design application developers who have tried to use existing database and programming language approaches.

2.1.4 Degree of Integration

One of the criteria for a *useful* extensible system is the successful integration of data definition, data manipulation, and general-purpose programming languages. Ideally, the users of this system should have a single language for expressing each of these aspects. The same mechanism which defines and manipulates objects in the database is available to define and manipulate the description of the database itself:

- Abstract description (properties and operations)
- Code to implement the operations (methods)
- Results of computation (exceptions)
- Storage representation (pragmas)

Extensible representations allow the possibility of run-time interpretation of the data structure and other elements of the schema. This run-time extensibility may require building self-describing structures at run-time. Since the surrounding structures used in these descriptions are not static, the database should provide a procedural implementation of properties and operations defined over objects in the database. Therefore, the code which implements these procedures must also be part of the database (and therefore also available at run-time). This integration permits a great deal of flexibility, and provides opportunities for optimization.

2.1.5 Efficient, Persistent Storage

Computer-aided design places two distinct demands on performance: it generates a great variety of data, and it generates a great volume of data. The system should provide mechanisms for efficient storage and manipulation of large textual databases through the ability to store generic objects of any size and to implement efficient, type based retrieval methods. Similarly, these same abilities should allow efficient storage and processing of non-textual data. For example, satellite images are often large in their original format (1-20 megabytes of raw data).

Developers must be able to dynamically create types and store instances of those types in the DBMS. Among the more challenging data storage requirements for data handling multiple versions of data objects and large, variable-length objects.

2.2 Past Approaches

This subsection describes some of the problems with the mechanisms we use today to provide extensibility in persistent storage. Programming Languages and Data Base Management Systems are two approaches to the problem. They have different, complementary advantages and disadvantages.

A *programming language* is the notation used to specify an implementation of an algorithm which can be executed by a machine.[Row80] A *Data Base Management System* (DBMS) is a general-purpose tool that accommodates the logical structuring, physical storage, and control of data; and that provides access interfaces to databases.[McL80] A DBMS manages data by providing mechanisms for data definition and manipulation, including integrity constraints and associative retrieval. Traditional DBMSs allow data sharing among several users, including data dictionary services. They are organized to efficiently store large amounts of data, and to guarantee data will not be corrupted due to system or media failures.[Ber87]

2.2.1 Expressibility

The necessary first step towards providing extensibility is being able to express it. The builder of an engineering design support application defines a set of types which models the entities of interest to the application. These types are defined as *subtypes* of the built-in types, and so they automatically *inherit* the basic structuring behavior defined by the built-in types.

Database models have historically focused only on the properties: they provide only a generic set of operations (*Join, Project, Select*) which operate on the containers (*records* and *tuples*) which are the only constructs the database provides for modeling real world objects. Programming languages have taken the opposite bias, modeling everything as an operation. Programming language systems such as Smalltalk[Gol83] or Flavors[Wei81] tend to force the type definer to specify operation triples (*get*_foo, *set*_foo, and *init*_foo) to handle what are really properties. Programming languages and databases have different notions of expressibility, and of just what kinds of things can be extended.

2.2.1.1 Programming Language An extensible language supports syntax, control-structure, data type, and operator extensions. A syntax extension allows the programmer to modify the syntactic structure of the language. Typically, the language allows the programmer to extend the data space by creating instances (Variables). In recent languages, the programmer can also express new data types.

Data Types New data types are specified in terms of a small, fixed set of type **constructors.** in some languages the semantics for built-in primitives (for example, *print*) can be extended to work on the new data types.

Relationships Programming languages allow the programmer to define relationships between objects using pointers. In Ada, these are called **access** types. Ada uses a different terminology to avoid the connotation of unsafeness usually associated with pointers. Ada pointers (that is, objects of an access type) are constrained to refer to objects of one type. Since access types can be part of **record** type definitions, the description of relationships can be extended from individuals to types.

Operations A control-structure extension changes the language control structures. An operator extension defines the semantics of an operator designation when applied to user-defined data types. For example, the plus operator ('+') could be defined to be concatenation when applied to two string values. Data type and operator extensions can be used to implement the abstract entities and operations used in an algorithm.

Data structuring facilities mirror those of operations:[Wir76]

TABLE 2-1. Correspondence of Program and Data Structures

Construction Pattern	Program Statement	Data Type
Atomic element	Assignment	Scalar type
Enumeration	Compound statement	**record** type
Repetition by a known factor	**for** statement	**array** type
Choice	conditional statement	**variant** record, type **union**
Repetition by an unknown factor	**while** or **repeat** statement	Sequence or file type
Recursion	**procedure** statement	Recursive data type
General "graph"	**goto** statement	Structure linked by pointers

2.2.1.2 Database Modern database systems are based (directly or indirectly) on the relational data model. In a DBMS, databases are generally limited to "structured" or "formatted" data. The data are logically organized in the form of discrete **records,** each containing a specified number of atomic **fields.** A DBMS assumes that the database it manages is considerably larger than its description. That is, a database contains a comparatively small number of kinds of data, and many instances of each kind. In contrast, *knowledge bases* typically contain many types, and one or a few instances of each type.

To the extent that traditional databases have supported extensibility at all, they have done so by allowing only a limited set of extensions to a particular aspect of an otherwise fixed "vocabulary." Normally, the only modification that can be made is to extend the database schema. This introduces new pieces of information, but does not allow new *kinds* of information; nor does it allow new ways of representing, accessing, or manipulating information.

2.2.2 Encapsulation

The term encapsulation is used here to include a number of related concepts — partitioning, decomposition, modularity, packaging, etc. The need for encapsulation arises from the complexity inherent in large software systems, which can only be handled by the application of systematic approaches to software development and the utilization of appropriate constructs supported by languages.

Partitioning should allow some division of a system into modules, with clearly defined interfaces between them. Modularity provides numerous benefits, such as easier maintenance, performance improvement, and enhancement, since only small, understandable parts of the system need to be understood and altered at any one time.

2.2.2.1 Programming Language The first modularity construct was the closed subroutine, which was soon recognized as a useful structuring device and generalized into the procedure mechanism of Algol 60. The synthesis of concepts from modularization, information hiding,[Par72, Par76] and abstraction generated the idea of using modules for supporting encapsulated data objects, or *abstract data types*.[Lis74] The first suitable construct was the *class* of Simula,[Dah66] later refined in Smalltalk[Gol83] and C++.[Str86] The **clusters** of CLU,[Lis77] **modules** of Modula,[Wir77] and **packages** of Ada[DOD83] implement the same kind of modularity.

2.2.2.2 Database A fundamental concern of database abstraction focuses on *representation independence*. The desire to separate the meaning of data from its computer-oriented representation has given rise to the notion of (physical) *data independence*. A database model is a formalism for expressing the logical structure of a database, and for providing a basis for manipulating such a database. The actual representation is separately specified, at a lower level of abstraction. Specifically, a database model consists of four logical components:

1. a data space, which consists of atomic *elements* and certain *relationships* among them,
2. type definition constraints, which specify restrictions on the relationships in the data space,
3. manipulation operations, which allow elements to be created and destroyed, and their relationships modified,
4. a predicate language, which allows individual elements to be identified by their logical properties and selected from the database.

2.2.3 Flexibility

By flexibility, we mean ease of use for the design application developer. Primarily, this means the facilities available to gain access to extensions as they are introduced.

2.2.3.1 Programming Language Modern programming languages have complex name spaces. The number of identifiers in large software system for subprograms, variables, types, constants, and so on is quite large. A fundamental mechanism for controlling this complexity is the concept of *name scope*. In Algol, Pascal, and later languages, name scopes may be nested in a *block structure*. Appropriate use of block levels helps control the visibility of identifiers. Languages such as Modula, CLU, and Ada provide individually named contexts. This provides greater control over the visible identifiers in a program's name space.

Languages with *allocators* also provide flexibility over individual objects. Normally, space for a variable is generated during elaboration of it's declaration. With dynamic allocation, the software explicitly allocates space for an object at run time.[1] Dynamic allocation gives programs great flexibility in their use of main memory space.

A final area of flexibility in programming languages is the use of *parametric procedures*. That is, the ability to pass subprograms as parameters to other subprograms. The using subprogram may invoke a different actual operation to carry out a specific operation. This is an instance of operation abstraction, as different actual routines may be invoked at different times to carry out the same abstract operation. The decision about which actual routine to invoke can be delayed until run time.

2.2.3.2 Database Databases provide extensibility by schema compilation. Most DBMSs provide access to the extensions from a programming language interface. Obviously, their flexibility is limited by the restrictions imposed by the Data Base Administrator.

DBMSs will sometimes provide a few different mechanisms for implementing the same feature. They may provide these facilities for users to select. For example, a DBMS may provide more than one access method. Users may select an indexed sequential or inverted index scheme for their data, depending on the expected usage. However, this sort of flexibility is usually fixed at schema creation time, rather than changeable at run time.

2.2.4 Efficiency

By efficiency, we mean those facilities provided to a design application developer to specify or influence DBMS performance. These can be for space and/or time efficiency.

2.2.4.1 Programming Language In most programming languages, the concepts of representation and storage are blended together in the concept of data structure - that is, representations always imply storage layouts. Usually this is a boon to the programmer, because the exact storage layout implied by the representations used are not important with respect to the semantics of his/her implementation. Occasionally, however, alignment criteria will force the representation and its implied storage out of sync. In these situations, the programmer has no recourse other than to alter the representation into one which generates the required storage layout. Some of the symptoms of this condition are obvious and not particularly debilitating: variables named unused, sparebits, or padding. Other times, the distortion introduced into the representation can be quite severe, as shown in this C code fragment:

```
struct PERSON
{
  char sex;
  long SocialSecurityNumber;
} People [1000000];
```

In this example, a PERSON is represented as a character named sex and a long (32-bit) integer named SocialSecurityNumber. Because of alignment requirements for long integers, the storage layout for this representation typically contains a 3 byte "hole". When multiplied by a million people, this may become a large enough waste for the programmer to worry about. Yet, in a language such as C, the mechanism for removing the hole is to change the type declaration of SocialSecurityNumber (say char SSN[4];). Changing the *type* of an entity is tantamount to changing its *meaning*. Thus it becomes impossible to have the desired storage layout and the desired meaning at the same time.

2.2.4.2 Database Knowledge-based systems are databases with more powerful front ends for dealing with the meaning of data.[Sow80] The primary difference between a traditional DBMS and an AI system lies in the volume of data that they process and the complexity of the representations. Databases still tend to have a large amount of repetition of very few types. In a DBMS, thousands of employees records may all have an identical format: one set of descriptors is sufficient to describe every record. In typical AI systems, however, the ratio is almost one-to-one: since there is so little repetition, each item must have its own descriptor. In fact, most AI systems don't even distinguish data items from data descriptors.

---

1. In so doing, the program takes responsibility for de-allocating the space. One of the difficulties with this scheme, not unique to languages, is the *dangling reference* problem — generating multiple references to an object, deallocating it, and forgetting to remove one (or more) of the references. Using the referenced space leads to unpredictable (and often disastrous) results.

In a conventional DBMS, each transaction reads records into its local workspace, operates on them, and writes modified records back into the "central" database. When the transaction terminates, the workspace disappears. Usually, the transaction executes for a fraction of a second, and rarely for more than a few hours. If the system fails while the transaction is executing, the transaction is aborted; and the user is given the opportunity to re-execute the transaction from the beginning.

It is inappropriate to regard each activity of an engineer in a design environment to be a transaction in the above sense. An engineer reads data into a workspace, and may operate on the contents of that workspace for many days. If the system fails during that period, it is unacceptable for all work on that workspace to be lost. Moreover, the work of two designers may not be serializable. They may work on shared data that they pass back and forth in a way that is not equivalent to performing their work serially.

2.2.5 Persistence

An important aspect of design database support, obviously, are the features provided to manage the data on secondary storage. In particular, we are concerned with the ability to retain information between activations of design application processes. For extensible systems, we are also interested in the ability to *incrementally* add to persistent storage.

2.2.5.1 Programming Language Programming languages rely on custom input-output operations for persistent storage of an application's data. The obvious difficulties introduced through duplication and wasted effort are usually justified on the inability of traditional DBMSs to support design needs with reasonable performance.

The Input/Output (I/O) facilities of programming languages use the semantics of the Operating System's file system. Ada provides some pre-defined packages for I/O operations, but these are clearly inadequate to the task of managing a large amount of data with sophisticated structuring and indexing needs. Managing persistent storage is really the forte of a DBMS.

2.2.5.2 Database Conventionally, relational DBMSs are designed to store small objects (namely records) and sets of small objects (namely files). Often the system has a small maximum length for either records or fields, which makes it impossible to store a large object as a single record.

A popular and important feature of virtually all DBMSs is the ability to retrieve data objects based on their contents. Content-based retrieval is valuable for many types of design tools. For example, a debugger may want to find all programs that modify a particular variable; a configuration management tool may want to find all modules that are checked out by a particular programmer; a project management tool may want to find all unfinished modules blocking completion of a particular release.

In today's record-oriented DBMSs (that is, relational, network, and many inverted file systems), there are four main considerations in implementing content-based access:

- Selecting index structures that map each value of a field into the records that contain that value. Currently, variations of B-trees and hashing are the most popular structures in DBMSs.
- Clustering records, so that all records with a given field value are grouped on a small number of disk blocks.
- Implementing set-oriented operators that efficiently retrieve a set of records that contain given field values (for example, relational selection) or that have field values that appear in other records (for example, relational join).
- Implementing an optimizer that transforms an algebraic expression of set-oriented operators into an equivalent expression that executes more efficiently.

There are several ways to modify the notion of a transaction to suit the needs of a design environment. One way is to ensure every transaction is short — so short that the abort of a transaction is no more than a minor inconvenience (for example, check-in or check-out). Longer activities (for example, fixing a bug, adding a feature) may consist of many transactions.

2.3 Criteria

Based on the analysis of the previous section, we can identify the basic facilities an extensible database should provide:

- expression of new data structures and data storage layouts
- incremental extension of properties and behavior, ideally as refinements of existing descriptions
- improved response time for gaining access to new database structures
- access to database extensions (by existing applications and/or utilities) without recoding, and in some cases without recompilation

TABLE 2-2. Summary of Design Needs

Criterion	Design Support Needs
Expressibility	Express complex modeling through objects, types, operations, and relationships. Enforce semantic restrictions through *strong typing* on properties.
Modularity	Separate specification from *implementation* of behavior. Cluster properties and operations into *abstract data type* descriptions.
Flexibility	Be able to dynamically create and access objects, types, operations and relationships. Handle exceptional conditions.
Efficient Persistence	Separate specification from *representation*, express compiler and storage **pragmas**. Rely on *automated* storage management facilities, which include concurrency control, associative retrieval, etc.

Table 2-2 summarizes the needs of design applications. We have focused on four areas of support for extensibility. These are extension **expressibility**, **modularity**, dynamic (run-time) **flexibility**, and **efficiency**.

2.3.1 Expressibility

We need to support extension not only to the information content of a database, but also to the kinds of information stored (including the representations for that new data). The goal here is to unify the features of **records** for example, random record access and fast compiled access code) with those of **tuples** (for example, indexes on tuples and dynamically defined tuple types) with good performance relative to the features used. We want to provide uniform access to data structures, and avoid having programmers continually re-invent the facilities to type, structure, index, link, robustly store, concurrently access, and cache data stored in primary and secondary memory. This is accomplished by extending the notion of a database schema to include a description of the storage layouts, access paths, manipulations, etc, which the system uses to implement a given function. The user is given access not only to the "external" functions supported by the database system, but also to the features which support those functions. The system's "meta-schema" then becomes extensible in the same way that an external schema is extensible. This allows extension to any part of the system.

2.3.2 Modularity

The impact of a given extension to the database should be clearly delimited and isolated from semantically unrelated areas. The question of how information should be localized within the system — of who should know what — is of critical importance to any large or complex application. A modularized extension reduces the *incremental* effort of making extensions, and allows the system to perform certain functions automatically (for example, recompiling affected modules).

2.3.3 Flexibility

There are alternative approaches to providing extensibility in databases: **Off-line**, as in traditional Database Administration schema changes (or a manual/batch database load/unload facility). The time scale for this operation (from request to completion) is measured in days. **Compile time**, for example special Type definition transactions. This reduces the time scale for application availability to minutes or hours. **Run Time**, such as incremental type definition or object creation code embedded in the application. The time scale here is in seconds.

Existing DBMSs provide compile-time extensions (via schema compilation). To meet the response time and access requirements, these extensions should be available as soon as they are defined (at run time). This implies application-level run time access to the database schema.

In the highly volatile environments typical of design applications, it is often unacceptable to take the database off-line in order to specify an extension. By dynamic extensibility, we mean the ability to extend the database without significantly impacting the performance of other, on-going applications. Therefore, in our approach to this project, we will focus on compile- and run-time methods.

2.3.4 Efficiency

Large applications invariably have areas which are performance or resource critical. If processing an extension to the database causes these areas to fall below a certain minimal level of acceptance, then the extension might just as well be inexpressible. It is necessary, therefore, that extensions are not "pasted on," but are incorporated directly into the system at the same level as the initial system features are incorporated.

The DBMS must cope with different representations of atomic types. The differences may be matters of machine architecture (for example, byte-ordering), programming language (for example, representation of strings), or tools (for example, representation of trees). The DBMS must know the representation of each type as it is stored and as it must be presented to users. The main efficiency consideration here is reducing memory-to-memory copying of data.

A DBMS must be able to store large, variable-length objects, such as documents and programs. Some large storage objects that are today stored as a single unit should be decomposed into smaller pieces, to take full advantage of DBMS facilities. For example, one could store each of a program's procedures together in a single object, as is typically done with file systems.

To avoid losing the partial results of a long-running transaction, it is desirable to *checkpoint* the activity periodically. Higher level, nested, transactions may be useful. Should one of the subtransactions fail, the entire transaction is not aborted. It may be useful for the DBMS to maintain a tree-structured history of the transaction and its subtransactions. The DBMS may allow the user to abort incomplete transactions, and undo or redo completed ones as a form of *backtracking* under the user's control.

The efficiency of the extension should be no less than the efficiency of the underlying database features; nor should it be any less than could be achieved by the implementor alone, without the database system. If we can develop a single language, and implement an extensible database using one language processor, we should be able to gain an additional benefit from strong typing: the ability to optimize run-time performance from compile-time analysis.

Our goal is to combine the expressibility of modern programming languages and the persistent storage management of databases with the inheritance/refinement mechanism of object-oriented systems to provide an extensible database system. In the next section, we examine the fundamental concepts underlying the successful features of databases and programming languages which support design applications. We also explore the unique features which object-oriented systems contribute to an effective solution.

3. Concepts

Figure 3-1 summarizes the discussion of the last two sections. Developments in programming languages, knowledge bases, and database systems are trending towards the same result: object-oriented technology. The inadequacies of existing technology is the driving force behind this trend.

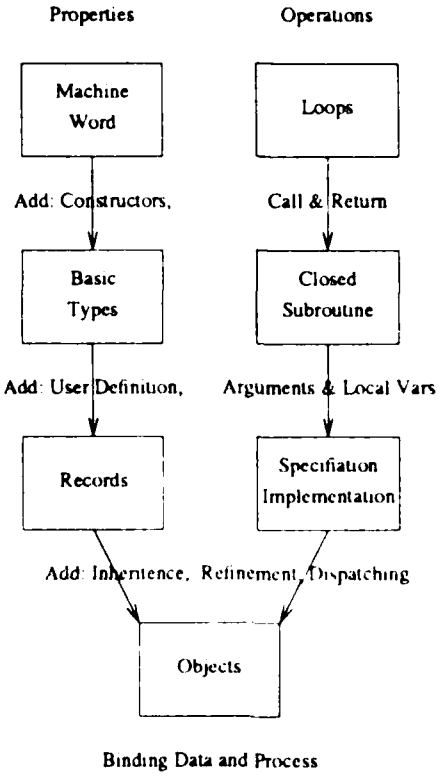


Figure 3-1. The Trend Toward Objects

This section describes the successful concepts these approaches bring to support of design applications, such as abstraction and data types. This section also introduces object-oriented systems, and the qualities which make them particularly attractive for extensible database systems. It concludes with a summary of the technical features which hold promise for providing support for an extensible DBMS. Our goal is to integrate these features in the UNIVERS product.

3.1 Abstraction

The way people normally deal with complexity is through **abstraction.** All languages provide an abstraction of a machine. *Control abstractions* are provided to specify the sequencing between statements in a program (for example, conditional branches) and *data abstractions* are provided to specify the entities and their operations used in the algorithm (for example, arrays with selection and assignment of individual elements).

A control abstraction provides the ability to extend the base language with new operations. Data abstraction is a mechanism which focuses on certain features of data and temporarily ignores other features, for the purpose of simplifying complex information. Data abstractions extend the basic types with new types. The ideas described below, which form the central core of the motivation for the Object Manager, are general enough to be of use in the construction of almost any application environment. They are of particular use in design environments, because of

the complexity of the design task. The more complex a data set is, the more it can benefit from the simplifying effects of data abstraction.

The data abstraction principle is that calling programs should not make assumptions about the implementation and internal representation of the data types that they use. Its purpose is to make it possible to change the underlying implementations without changing the calling programs. A data type is implemented by choosing a representation for values and writing a procedure for each operation. A language supports data abstraction when it has a mechanism for bundling together all of the procedures for a data type.

Parameterized data abstractions define a hierarchy between types. The parameterized abstraction is a generalization of an instance of the abstraction which has a particular value bound to the parameter. Note that type parameterized data abstractions usually have some generic procedures.

Exceptions are types which enforce simple and consistent handling of errors and other unusual circumstances. Operations and their methods are designed to perform certain tasks. However, in some cases a method's task may be impossible to perform. In such a case, instead of returning normally, which would imply successful performance of the task, the method should notify its caller by raising an exception. They are primarily a mechanism for communication among programs. A failure occurs when the abstraction is "broken" and no longer meets its specification.[Lis86]

Iterators provide a generalized looping construct and are used to examine each element of an aggregate, one at a time. Aggregates include all collections of entities, such as arrays or sets. The iteration abstraction is a generalization of the iteration methods available in programming languages.

Conventional commercial database management systems support only one type of abstraction -- *instantiation* (**an-instance-of**). In a CODASYL system, a record is an instance of a record type; a particular set is an instance of a set type. In a Relational system, a tuple is an instance of the type defined by its containing relation.

Design applications also use *realization* (**an-implementation-of**) and *association* (**a-member-of**).

Artificial Intelligence knowledge representation systems have for many years incorporated two other basic abstractions:

- *generalization* (variously referred to **a-kind-of** or 'is-a'), and
- *aggregation* (**a-part-of** or 'consists-of')

We focus here on three kinds of abstraction: **generalization, aggregation,** and **instantiation.**

3.1.1 Generalization (*A-Kind-Of*)

Types are related to one another in subtype/supertype hierarchies. The complementary notion is *specialization*. Given a pair of hierarchically related types, the more general one is termed the *supertype* and the more specific one the *subtype*. What gives this notation its power is that it serves as the base for an automatic inheritance mechanism. The designer need not go through the drudgery of defining common properties on each subtype in the hierarchy. This reduces the specification task and also allows a significant increase in modularity of representation.

3.1.2 Aggregation (*A-Part-Of*)

The a-part-of abstraction is ubiquitous in design support applications. Things are made up of other things. Design is in fact often a process of specifying the subcomponent structure of a complex artifact, using the complementary concept of *decomposition*.

Aggregation shows up in many different forms. It is explicit in the ability to make groups of objects. These groups can be formed either by inserting their members manually (just as one might explicitly make a grocery list by writing down each item); or by forming a *predicate* which describes all the instances (for example, the group consisting of all blue-eyed professional football players). These aggregates can exhibit a variety of behaviors, such as an ordering of the elements in the group, an index over the elements, etc.

Aggregation also appears in the definition of the properties of an entity (for example, people have friends). We might think of Friends as a property which relates one person to a group of people (his friends). Or we might think of there being a group of Friend properties, each of which relates one person to one other (his friend). In either

case, the group might be ordered (by how close each friend is), or indexed (by the friend's name), etc. Groups, sets, lists, arrays, etc, are all examples of aggregates.

3.1.3 Instantiation (*An-Instance-Of*)

This concept is also called *classification*. The class of all entities is partitioned along two orthogonal lines into:

1. *types* and *instances*
2. *objects*, *properties*, and *operations*.

We say that entities are instances of *types*. Types define the properties carried by each of their instances and the operations which may be executed on each instance.

The usefulness of data abstraction constructs depends on reasonable type-checking rules. *Name equivalence* defines two values to be type compatible if the "names" of the types are the same. *Structural* equivalence defines two values to be type compatible if the data representation is the same, regardless of how it is declared. Type constructors can be type-checked using structural equivalence and named types can be type-checked using name equivalence.

Both abstraction-based and object-based languages have addressed the importance of *representation abstraction* — separating behavioral specification from its implementation. Without dwelling on the advantages of this separation, it will be useful to define loosely what we mean by these terms. Specifications are a formal way of saying how something should be used. Implementations are a formal way of saying how something should be built — so that it meets its specification.

Specifications and implementations are closely related. Specifically, the implementation of abstract objects rests on the specifications of other abstract objects. An implementation consists of an *active* and a *passive* part. The active part consists of methods, and the passive part, called representation, consists of data structures. Methods, representations, implementations and abstractions are related in that a method manipulates a representation (by using that representation's specification) in order to implement an abstraction.

3.2 Types and Behavior

An *entity* is any separable and identifiable piece or nexus of information. An entity is anything to which one can refer. Examples in the real world include: a tree; a leaf of a tree; a color; the concept (type) "tree"; a particular species of tree; an operation on a tree (for example, cutting it down); a piece of wood; a group of pieces of wood; a table; a dinner party; a person; a relationship between two people.[2]

In general, the only things that must be true of all entities are: they are instances of some type, and they have *behavior*. If something has no types, then it has no behavior. If it has no behavior, then it has no distinguishing characteristics: it cannot be distinguished, and from the point of view of the Object Manager, does not exist. If something does have behavior, then it can be observed, potentially manipulated, and classified according to that behavior. We define behavior in terms of the properties which an entity has (or can have), and the operations which can be applied to it.

The nexus of information about behaviors common to a group or class of entities is called a *type*. The role of a type is to keep track of and manage a group of behaviors. Every entity which has that group of behaviors is said to be a member of the class of the type, or simply an *instance* of the type. The type manages (keeps track of, implements) the behaviors which it aggregates, and is therefore sometimes called a *type manager*. An entity can be an instance of more than one type, and it gets behavior from all of its types. Each type then has a partial hand in implementing the behavior of the entity.

---

2. Note that an entity need not have a physical existence: a concept is still taken to be an entity. An entity can be transient, as in the case of an operation or action. In the terms in which the Object Manager deals with entities, an entity can either stand for something in the real world, or it can be purely synthetic, that is, it might stand for nothing other than itself.

A type defines the behavior of its *instances*. Every instance of a type must obey that type's *behavior specification*. The type guarantees compliance not only by specifying its instances' behavior, but also by implementing that behavior. There are two major pieces to any type definition: the type's behavior specification (or simply its spec) and its implementation.

Not only objects, but also properties and operations, have types which describe their behavior. These types are first-class entities, which are denotable in the language, and can be manipulated just like any other entity. Because the behavior of operations and properties are described by operation and property types, it follows that there are operation and property subtypes.[3] One of the most serious drawbacks of the Smalltalk class of object systems is their lack of any notion of type specification. There are simply objects.

Early object-oriented programming languages like Smalltalk were constrained by a loose notion of subtypes and a commitment to late binding. When an object invoked an operation or referenced a property, the interpretor followed a pointer from the object at hand to its type defining object. It then looked up the operation selector in a dictionary (and/or repeatedly went from the type defining object to its supertype). Finally, it set up a context for the method and eventually executed its code (again interpretively). This applies both at the instance level and the data structure level (in Smalltalk: instance variables).

The benefits of strong typing are well known:

- strong typing resolves many more errors at compile time. Compile time errors are generally easier to analyze and correct than errors of type mismatch that occur at run-time.
- strong typing of object data structures provides superior specification of the system. Rather than relying on naming conventions of programmers to convey type, data structure type declarations provide a clear specification.
- A strongly typed system allows the language processor to do far more analysis at compile time. This analysis can eliminate the need for run time checks. For object systems, this analysis allows methods to be statically bound as well. This eliminates a large amount of the performance overhead normally associated with object systems.

3.3 Object Oriented Systems

Over the last decade or so, software research has been converging on a very powerful model called the object-oriented or simply the object model. The term object-oriented programming has been used to mean different things, but one thing these languages have in common is *objects*. Objects are entities that combine the properties of procedures and data, since they perform computations and save local state.[Ste86] Uniform use of objects contrasts with the use of separate procedures and data in conventional languages.

More generally, an object can be thought of as an abstraction of some "real world" entity. It is thus a good paradigm bout to express the problem and its solution. It turns out the object model is a good way to model both real, external objects in the application and internal implementation objects in the computer.[4]

Object-oriented programming emphasizes the view that a program largely describes the definition, creation, manipulation of, and interaction among, a set of *well defined* and *independent* data structures, called objects. Most of the object-based formalisms are essentially based on the same fundamental concepts:

Objects Objects are entities which contain state information. Each object has some predefined attributes or *properties* which collectively form the internal state of that object. In addition, for each object certain actions or *operations* are defined. An operation performed on an object may change its

3. Types are related by subtype/supertype properties. These properties structure the space of types into a directed acyclic graph (DAG).

4. A software design approach with the same emphasis on real-world modelling can be found in the Jackson Structured Programming and Jackson System Design methods.[Jac75, Jac83]

internal state (the value of one or more of its properties). An operation may also cause other operations to be invoked on other objects.

Types Types define a collection of objects with similar behavioral properties. The behavioral properties of an object refer to the various possible states of the objects (that is, the properties) and the possible actions that can be performed on a given object (in each state).

Instances An instance of a type refers to a concrete object, which belongs to a given typ. An instance of a type is created from the type definition. Once an instance is created it may behave as specified by its type definition.

Inheritance Types may selectively derive some of their behavior (properties and/or operations) from one or more other types. This mechanism is called inheritance, and it applies only to types. Any instance of an inherited type inherits all its behavior from its "supertyp(es)".

Defaults Some initial state which characterizes all objects in a given type may be specified by a default value mechanism. If a default is specified for a particular type, then all instances of that type are created with a state with the default values selected.

3.3.1 Objects

The object model is based on the **object** an entity that has both static state and dynamic behavior. The object is a good model in the problem space because it, like the real objects it models, can express static and dynamic properties. The object is also a good model in the solution space because it encapsulates both static state and procedure into a single programming module.

An object is an autonomous module. It has its own data and procedures so that it can model both static and dynamic behavior. It is accessed through a well defined interface and appears to the rest of the system to behave just like the real object it models. In this respect, object-oriented system resemble the **abstract data type** approach in programming languages. The other information associated with an object, for instance, its representation) is hidden from the users of the object definitions.

Collectively, an object's properties and operations are called its *behavior*. The term *behavior* is used to underscore an object's dynamic nature. Even property access is dynamic since it must be made via one of the object's operations.[5]

3.3.2 Polymorphism

There is additional leverage for building systems when the operations are standardized. This leverage comes from **polymorphism.** Standard protocols enable a program to treat uniformly objects that arise from different types. These protocols extend the notion of modularity (reusable and modifiable pieces as enabled by data-abstracted subroutines) to polymorphism (interchangeable pieces as enabled by method dispatching).

Another important concept in object systems is **specialization.** Specialization is a technique that uses type inheritance to elide information. Inheritance enables the easy creation of objects that are *almost* like other objects with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place.

Specialization and method dispatching synergize to support program extensions that preserve important invariants. Polymorphism extends downwards in the inheritance network because subtypes inherit protocols. Instances of a new subtype follow exactly the same protocols as the parent type, until local specialized methods are defined.

---

5. Some object systems use *message passing* to communicate with objects. Message sending and method dispatching are both forms of indirect procedure call — with different implications for performance.

3.3.3 Inheritance and Refinement

Certain features of Object-Oriented systems make them particularly well suited to fulfill the needs of extensible databases. One of the fundamental ideas in object-oriented systems is the use of *inheritance* in type hierarchies. This style of inheritance has been useful in defining semantic network models for Artificial Intelligence applications, and for semantic data models in database design. Inheritance is one of the key concepts in object systems. It not only makes writing type definitions easier, but it determines many of the desirable dynamic properties of object systems.

The objects of the application — machine parts, paper work objects like Engineering Change Orders, orders, materials transfers — are modeled by system objects. Objects are, in turn, defined by object type definitions. All objects of the same type share a common definition. And types are hierarchical. They form a tree with the most general type at the root. If we were modeling fastener hardware, for instance, the most general type may simply be called Fastener. At the next level, it is refined into a number of categories — say Screws and Nails. These, in turn are refined further. At each level the type inherits all the behaviors of its parent (called *supertype*). It adds to them and passes the composite behaviors to its *subtypes*.

An object can represent a software construct as well. It might represent a data entity like a record or an array and include the procedures for verifying it or expanding it from compressed storage. An object might be a shared library procedure and include not only the code but also parameters, table values and other static information. For instance, procedures for dealing with sorted lists might be encapsulated into a single object. The object's static information would store the collating sequence; its dynamic procedures would include such list functions as sorting and inserting.

The nexus of information common to a set of types is called the *supertype*, or generalization of those types. The lower level types are called *subtypes*, or specializations of the supertype. The role of a supertype is to factor out common behaviors defined by other types. Every instance of one of the subtypes will automatically be given all of the behaviors defined by not only the subtype, but by the supertype as well. We say that the subtype *inherits* the behaviors defined by the supertype. It is exactly as if the subtypes all defined these behaviors themselves, except that they only get defined once, thus simplifying the specification task and increasing system modularity.

Types are generalized and specialized by the supertype/subtype relationship. When instances are classified by type according to their behaviors, there is often quite a lot of overlap between the types. For example, we might note that broad-leafed trees and conifers both have trunks, photosynthesize, etc. but that they differ in that conifers have needles, not leaves, and that they reproduce via cones, rather than seeds. Generalization furthers the structure of types imposed by classification by introducing the notion of a lattice of subtypes and supertypes.

Normally, supertypes are introduced to aggregate common behaviors of a set of types. For example, cars and motorcycles both have Enginesize properties and Refuel operations. A supertype, called MOTORVEHICLE, could be introduced for the central definition of these behaviors. The subtypes, CAR and MOTORCYCLE, inherit the behaviors defined by MOTORVEHICLE, and do not need to define them directly.

The behaviors defined by a supertype may be augmented by new behaviors defined by a subtype. For example, CAR defines the Drivewheels property, which has value **oneof** (*front, rear, all*); and MOTORCYCLE defines the Starter property, with value **oneof** (*kick, electric*). If a subtype does nothing but add new behaviors to those defined by its supertypes, then we call it a purely *additive* subtype.

We also allow a subtype to refine, and even constrain the behaviors defined by its supertypes. For example, BSA motorcycle (a subtype of MOTORCYCLE) refines the Starter property to only have value *kick* — because all BSAs have kick starters. Both property and operation behaviors can be refined by subtypes of the type which defines them.

A subtype typically either defines new behaviors (properties or operations), or changes the definition of behaviors defined by its supertypes. If a subtype changes the definition of an existing property or operation, then this change must be made compatibly with the supertype's original definition. Such a change is called a refinement of the original property or operation.

In effect, each type defines a set of objects; its subtypes define particular subsets of those objects. Therefore, an object variable of a particular type can be used to hold objects of that type and of *all subtypes* of that object. An

object variable can be used to hold any of its subtypes in any context, including as an argument to an operation.[6] Furthermore, if the operation has been refined in the subtype definition, then dispatching to the refining operation is performed dynamically according to the actual type of the argument at runtime.

Type definitions are hierarchical and are *inherited* along the type hierarchy. Each type inherits all the property and operation definitions of its supertype. Then it refines them further in its definition, extending them or making them more specialized. If it has subtypes, it passes all its inherited and refined behaviors to all its subtypes. Thus type definitions are written incrementally, as additions or *refinements* of their parents' definitions.

The type hierarchy and the inheritance mechanism can be thought of as modeling the **a-kind-of** relationship between data elements. For instance, the *Screw* and *Nail* types discussed previously are each a a-kind-of *Fastener*. Similarly, a specific instance of the type *Screw*, say a one inch, round-head, number 6 screw, is a a-kind-of screw.

Properties are refined by adding new properties or constraining inherited properties. Operations refinement is implemented by enclosure. The refining operation in the subtype is called first. It, in turn, calls the operation of its supertype. In that way additional processing may be defined both before the supertype operation is called and after it returns. The supertype operation may also be called more than once (or not at all) depending on what refinement is required.

3.3.4 Polymorphism and Dispatching

Frequently, the "same" operations have to be performed on wide classes of objects. For instance, we might want to calculate flow through a complicated piping system. At some level of abstraction, flow can be expressed as a function of pressure. Unfortunately, in practice different algorithms are used for different kinds of flow elements. While flow may be calculated from directly from dimensions in simple elements it might be determined from empirical tables in complex ones. A similar situation is often true for internal operations. While multiplication may be equally meaningful for scalars, vectors, complex numbers and matrices, the implementation in each case may be quite different. Ideally, we would like to express the operations symbolically and let the system fit the implementation to the actual data types involved. Dispatching provides that exact behavior.

Dispatching is the runtime binding of an operation call according to the type of its principle argument. Dispatching executes the most refined or most specialized operation determined by the actual type of the argument passed at runtime. In effect, this allows making generic operation calls applicable to large classes of data and letting the system find the correct refinement of the operation according to the actual arguments passed.

Methods describe how an object will perform its operation. Operations specify their invocation and termination behavior; methods are the subroutines which do the work of the operations. When operations are invoked by an application program, the state and parameters are set up according to the invocation specification, and a subroutine call to the method is made. When the subroutine exits and returns to its caller, the state and returned parameters are handled according to the operation's termination specification.

When a subtype refines an operation of its supertype, it can either wholly replace the method associated with the supertype's operation, or add new behavior to the supertype operation's method by combining the subtype operation's method with the supertype operation's method. Method combination defines the interactions and flow of control between the various methods implementing a single operation and its refining operations. The Object Manager allows both trigger methods and base methods to be combined.

3.4 Summary

Table 3-1 summarizes the features used in these approaches to satisfy the criteria of Section 2.1. Programming languages are quite expressive, but lack automated storage management. They allow fine control over main

6. Since a subtype's definition cannot contradict the definition of its supertype, any operation that can be performed on a type can automatically be performed on all its subtypes.

memory storage, but little effective control over secondary store. Input-output facilities vary widely between languages, and even between implementations of the same language. Database systems manage storage efficiently, but have a fixed set of operators and data structuring methods. Database extensibility is awkward, usually via schema re-compilation (under the control of a Data Base Administrator).

TABLE 3-1. Approaches to Extensible Databases

Criterion	*Programming Language*	*Data Base*	*Object-Oriented*
Expressibility	Variables and types	Relations and records/tuples	Objects
Modularity	Subprograms and packages	(none)	Spec/Rep separation
Flexibility	Allocators	Schema definition	Inheritance, Refinement
Efficiency	Representations	Indexing, Clustering	Representations
Persistence	File I/O	Extensive	Clustering, storage pragmas

In the next section, we examine the issues surrounding a unification of these approaches. In particular, we consider the integration of the programming language Ada with the object-oriented database Vbase.

4. Issues

This section presents some of the issues surrounding the development of an Ada programming language interface to the Vbase object-oriented database. Our basic goal is to add (transparent) persistence to Ada — unifying the expressibility of that language with the database facilities of automated storage management. We discuss programming language issues, database access issues, and application run-time issues involved in achieving this unification.

Ontologic has been actively designing and developing an object-oriented database product since October 1985. and had been working with the technology since mid 1983. A complete implementation of a single-user (non-shared) object-oriented data management tool has been completed and is now undergoing evaluation with a customer-provided prototype design application as a test vehicle. This initial product will soon be extended to handle multiple users. At the point in time when it provides for the sharing of data, it will have become a database system. Our target for public release of the database product is early 1988.

In the process of building this product, we have had to develop not only the product itself, but also expertise in database system implementation, expertise in language processor design, and many support facilities. We have learned how to test extensible systems, how to keep them self-consistent, and how to deal with evolution.

We think that the resulting product will be applicable not only to design support applications, but also to a host of management and commercial applications ranging from Program Management of large development efforts to office automation, the management of financial portfolios, medical records and many many others. We believe firmly that the object-oriented model for information and its use will be the next great wave in computer software.

4.1 Vbase Features

The Vbase product consists of:

- A storage manager which relieves the user from a large portion of the effort of dealing with data structures for objects.
- A collection of Type definitions which provide the intrinsic behavior of the object-oriented system, such as extensibility, information hiding, etc.
- A Type Definition Language for extending the set of Types in the system. This facility is equivalent to the Data Definition Language of a traditional database. TDL is a database application and uses database operations to implement its functionality.
- An extended 'C' language through which objects can be manipulated, the Data Manipulation Language of the product. This extension is implemented as a preprocessor, and the preprocessor is a database application. It can produce highly efficient database access code.
- A set of utility programs for debugging extended 'C' programs, displaying and editing the database, etc.
- Soon it will also provide an object-oriented graphics facility built out of the capabilities of the basic database.

The Vbase product is a self-describing, object-oriented, software development environment, and a mapping of that environment into the world of the C programmer. It provides the C programmer with mechanisms for the definition of object-oriented data structures, with support for the persistent storage of the objects created, and with a strongly-typed extension to the C language for reference to stored objects.

As the first step towards providing a complete design software development environment, we created a database management system with two language interfaces, one for data definition (the Type Definition Language, or TDL) and one for data manipulation (the C Object Preprocessor, or COP). The Vbase system runs under UNIX 4.2 BSD on Sun 3 workstations. Figure 4-1 shows the basic components of the Vbase Release 1 system, and how they interact with each other. Type definitions of objects are compiled by the TDL compiler. Object functions and applications programs are written in COP and compiled by the COP compiler. When the application runs, it uses the run-time Object Manager to access data and invoke operations.

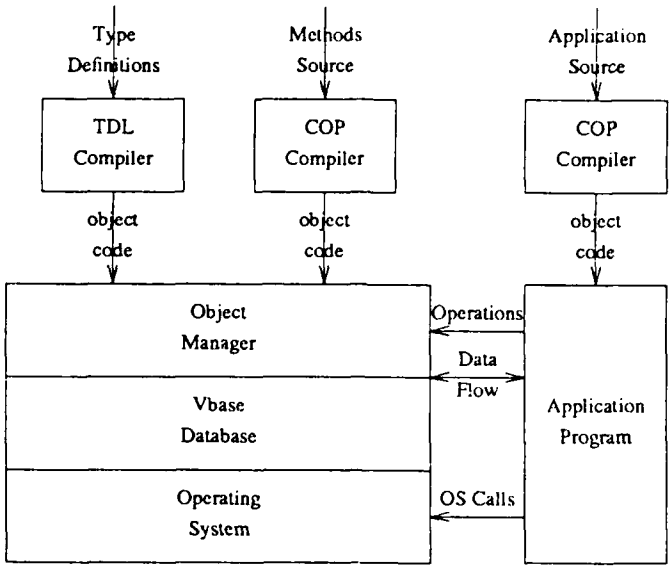


Figure 4-1. The Vbase System

When we speak about an "object-oriented" facility, we mean to include the following characteristics:

- The encapsulation of data descriptions and the code which operates on that data together to define object TYPES. Some developers, such as those of the language Smalltalk, have called these encapsulations Classes.
- The ability to create instances of a Type and to operate on the data of the instances (PROPERTIES of the instances) through the OPERATIONS of the Type. We call such instances OBJECTS.
- The system provides for properties which link two or more objects together by RELATIONSHIPS into complex objects.
- Users perceive the system as providing the ability to grasp and manipulate complex objects and object structures without regard to how big or complex they might be or how their internal representation might be done. Thus objects are in some sense abstract. It is equally easy to operate upon an INTEGER, a PICTURE or the design of a Boeing 747. The users need only know what operations are available for each Type, and the system warns them of potential errors via *type checking*.

4.1.1 Strongly Typed Languages

The language defines **entities. Types** are the most common entities. A type serves as the nexus for behavior of its **instances.** The basic components of a type are its **properties, operations,** and **supertypes.** Properties represent static behavior, while operations describe dynamic behavior. The supertype definition places the type within the type hierarchy. Behavior is inherited along the type hierarchy. A type is also a block scope, and may therefore contain other definitions along with its central property and operation definitions.

There is a taxonomy of types, with subtypes inheriting both properties and operations from their supertype. Subtypes can add more specific behavior by specifying additional properties or operations, and can also refine existing behavior. When an operation is invoked, it is dispatched according to the type of the object of the invocation.

The block structure of Vbase is different from most object-oriented systems, and certainly very different from most DBMS schema definition languages. It supports the kind of complex name environment of structured programming languages, with the same reduction of name conflicts. It also supports path names, allowing simple groupings of names, and relative naming.

An important feature is the ability to arbitrarily combine program variables and object variables. The language processing does all necessary conversions to preserve correctness.

4.1.2 Flexible Operations

In Vbase, operations are viewed as being implemented by a series of executable code fragments. Perhaps the most notable run-time feature of Vbase is method combinations. Method combination in object systems results when a refining method invokes its refinee. Smalltalk[Gol83], for example, provides the *super* pseudo-variable for this purpose. The number of fragments is arbitrary, and is the sum of all triggers and methods defined in the operations. Each operation is associated with a base method and an arbitrary number of triggers.

Triggers can be attached to properties as well as operations to generate "active object" behavior. These behaviors include the standard ones — "when my QuantityOnHand property gets below 20, issue a new order for 100 more — to more esoteric uses (such as keeping audit trail records of property and operation access for security purposes). Triggers are most often used to augment create and delete methods. This use of triggers allows the user to insure that upon creation of the object, all important semantic restrictions (such as all referent object relationships) are created properly. Delete triggers reverse this to delete all referent objects — thereby eliminating a major source of "dangling references".

Another feature is the **iterator.** Drawn from CLU[Lis77], this operation provides the ability to process all elements of an aggregate one at a time without having to manually program the proper indexing of the loop. This eliminates a major source of programming errors — "off by one" counts in loops.

4.1.3 Powerful Data Model

The Vbase Data Model is completely self-describing: all system characteristics except the lowest levels of storage management are implemented using types. The properties and operations of these system types are freely available to programmers to use to their advantage. The accessibility of meta level, or dictionary, information has long been a stamp of DBMSs. Object systems generally allow some access to this information, but Vbase goes further to allow virtually total access to system types. By providing users with access to the system name types, users can create named entities that will be recognized by system tools such as the debugger and object editor.

We say that the system is self-describing and extensible because the same mechanism which defines and manipulates objects in the database is available to define and manipulate the description of the database itself:

- Abstract description (properties and operations) via types.
- Code to implement the operations (methods).
- Results of computation (exceptions).
- Storage representation.

Type descriptions are objects. They too have a type description, provided by type TYPE. Since TYPEs are instances of a special type, they are not much harder to create than instances of other types from the point of view of the database. The operations which create them are OPERATIONS provided by database TYPE definitions. The only tricky part is making sure all of the pieces which each TYPE needs are in place and that the names needed to refer to them get put into the correct place, in the database of course. The Vbase compilers take care of all the details.

Types can be created as *refinements* of other, existing types. We call the refinements *subtypes* and the original a *supertype*. This provides Vbase with the inheritance mechanism characteristic of object-oriented systems.[7] In Vbase, properties and operations of the supertype become automatically available to instances of the subtype. This creates another form of extension to the database, and fosters re-use of code and incremental programming, which is very efficient. We made sure that simple instance access — often ignored in object-oriented systems — remained efficient and simple, so that traditional database access remained acceptable for design applications.

---

7. This model not only supports but requires complete support for **a-kind-of** hierarchies that the current literature agrees is necessary to define the complex data structures and relationships of engineering databases. We extended the model to support **a-part-of** hierarchies, which are essential to integrated, consistent design of component parts and their assemblies.

Exceptions [illegible] in Vbase, exceptions are types. This means that [illegible] is available for exceptions. The first [illegible] exceptions. This allows the generalization of [illegible] of exceptions as types is that [illegible] useful. One can consider the [illegible] instance is available to the exception [illegible] operations, etc.

[illegible] representations [illegible] the data structure and other elements [illegible] structures at run time. Since the [illegible] database should provide a procedural [illegible] database. Therefore, the code which [illegible] at run time.

4.1.4 DBMS Persistence

[illegible] shared among multiple processes [illegible] Persistence of objects is [illegible] At any time an object is [illegible] a fetch operation.

[illegible] storing, retrieving, and manipulating data.

- [illegible]
- high performance database [illegible]
- persistent [illegible]
- Graphics subsystem
- support for basic data types and [illegible] of data

All applications will gain from these features [illegible] whether they are object oriented or not, although the greatest benefit is intended for object oriented systems.

Unlike most database management systems, we did not build an invisible wall between Vbase and the users of Vbase: rather, Vbase opens the database to the users by

- providing the power of the database to users directly with Create, Delete, Get, and Set method refinement
- decoupling the data model from the internal workings of Vbase, so that the data model itself can be enhanced and extended to fit changing user requirements
- extending the Vbase data model with the user application code, with all of the performance gain, extensibility, and ease of use that that implies.

We made a radical departure from the database program environment split and incorporated the database *into the process memory* of programs to improve database performance so that our Graphics subsystem could give the support and performance required for CAD/CAM graphics processing. This solution has only become possible within the last few years as workstation manufacturers have dramatically increased the available MIPS and virtual memory of design workstations, while bringing down the average cost of workstations to make them widely available.

By embedding user application code as operations of Vbase, we opened the power of the system to users. At the same time, we made the task of developing Vbase through future releases easier by using its object-oriented data model to describe itself.

Another database facility is the ability to cluster objects. Every create operation allows the invoker to specify a clustering object. The new object will then be created in the same **segment** as the clustering object. Segments are the unit of transfer to and from secondary storage. Thus, a number of objects can be clustered, and whenever any

one of the objects in the cluster is accessed, all of the associated objects are available. Any subsequent references to one of the clustered objects will not require a disk access. Clustering also provides better space utilization of secondary storage.

An innovative feature of Vbase is support for **inverse** relationships between properties of different objects. This means that whenever a modification is made to one of these properties, the other object's property is modified accordingly. This construct solves one of the more vexing problems in DBMSs, particularly relational systems. One-to-one, one-to-many, and many-to-many relationships can be supported and maintained automatically using the inverse capability. Such common relationships as Parts-Suppliers of Employees-Departments can be implemented directly with no additional definitions or code. This is a dramatic improvement over most current database systems, and is not available in current object systems at all.

4.2 Ada Features

We will not attempt to describe all the features of Ada, but instead concentrate on those that relate to describing a Vbase system interface. We will evaluate the effectiveness of the interface by its support for extensibility, its "self-descriptive" capacity, and its consonance with existing Ada semantics.

4.2.1 Type Checking

Ada's emphasis is on *static* type checking. It is possible to continue to enforce strong type checking, even in the face of dynamic (that is, run-time) type and object definitions. Our approach is to integrate type checking in the pre-processor, doing as much static type checking as possible. We will rely on Ada's static type checking in the generated output, and on Vbase's dynamic type checking in the database.

Changes in the database which are (in this version of the product) independent of changes to the Ada compilation library. This presents redundant updating problems only to the extent that users attempt to maintain duplicate descriptions in the two. Our approach is to automatically generate the necessary Ada compilation units (for example, package specifications and bodies) as output from the pre-processor. This approach should minimize the Ada code that users will want to insert independently into the Ada library for database operations.

Vbase provides a mechanism for run-time type definitions. There is no equivalent for this in Ada, although Ada provides run-time *constraint* checking. Those users who take advantage of run-time type definition and use must rely on the Object Manager's dynamic type checking mechanism, not on Ada's compile-time checks.

4.2.2 Subtypes and Inheritance

Ada supports inheritance through the *derived type* mechanism. It has the following features:[DOD83]

- The derived type belongs to the same class of types as the parent type. The set of values for the derived type is a copy of the set of possible values for the parent type. If the parent type is composite (for example, a **record**), then the same components exist for the derived type, and the subtype of corresponding components is the same.
- For each basic operation of the parent type, there is a corresponding basic operation of the derived type. Explicit type conversion of a value of a parent type into the corresponding value of the derived type (and vice versa) is allowed.
- If a default expression exists for a component of an object having the parent type, then the same default expression is used for the corresponding component of an object having the derived type.
- If an explicit representation clause exists for the parent clause and if this clause appears before the derived type definition, then there is a corresponding representation clause (an implicit one) for the derived type.
- Certain subprograms that are operations of the parent type are said to be *derivable*.[8] For each derivable

subprogram of the parent type, there is a corresponding derived subprogram for the derived type.

- The specification of a derived subprogram is obtained implicitly by systematic replacement of the parent type by the derived type in the specification of the derivable subprogram. Any subtype of the parent type is likewise replaced by a subtype of the derived type with a similar constraint. Finally, any expression of the parent type is made to be the operant of a type conversion that yields a result of the derived type.
- Calling a derived subprogram is equivalent to calling the corresponding subprogram of the parent type, in which each actual parameter that is of the derived type is replaced by a type conversion of this actual parameter to the parent type. In addition, if the result of a called function is of the parent type, this result is converted to the derived type.

4.2.3 Modularity

Ada provides the classical subprogram facilities for extending operations — **procedure** and **function** similar to those of Algol 60 or Pascal. The Algol-like block structuring allows nesting to control scope and visibility. A major restriction is the lack of procedural parameters (that is, parameters which are themselves procedures or functions).[9] It is possible to write more than one subprogram with the same name, but with different types of parameters. This *overloading* of names is a version of compile-time *polymorphism*, which provides a great deal of flexibility.

Ada's separation between a subprogram *declaration* which defines its external interface, and its *body* which contains the implementation details, solves some traditional problems associated with procedural languages (for example, mutually recursive procedures). Ada provides the **package** construct so that logically related items can be grouped together. It permits the definition of both encapsulated data objects, and abstract data types.

As with procedures, the declaration and body of a package are physically separate in the program text. Only those names declared in a specification may be available outside the package. Thus for large programs, partitioning into packages provides a greater degree of control over the visibility of names than is possible with traditional block-structured languages. Access to the visible parts (that is, the specifications) of packages is not automatic, it must be explicitly stated in the "importing" unit. A greater degree of information hiding is available through the use of the **private** facility. This allows a type to be specified in a package specification and thus be accessible externally, but the representation of the type (in terms of other types) to remain hidden.

A further encapsulation facility available in Ada is the ability to specify a procedure, function, or package as **generic,** which allows it to have formal parameters which are **types** or subprograms. Particular units are *instantiated* from the generic units by substituting actual parameters.

Ada also provides separate compilation of program units. Separate compilation differs from independent compilation provided in traditional languages (for example, FORTRAN, C) in that type checking is carried out across the separate parts. Whether system components are assembled "top-down", "bottom-up", or in any order, the visibility and interface information (including typing) is enforced by the Ada program library.

---

8. Two kinds of derivable subprograms exist. First, if the parent type is declared immediately within the visible part of a package, then a subprogram that is itself explicitly declared immediately within the visible part becomes derivable after the end of the visible part, if it is an operation of the parent type. (The explicit declaration is by a subprogram declaration, a renaming declaration, or a generic instantiation.) Second, if the parent type is itself a derived type, then any subprogram that has been derived by this parent type is further derivable, unless the parent type is declared in the visible part of a package and the derived subprogram is hidden by a derivable subprogram of the first kind.

9. However, Ada's generic units allow subprograms as arguments, and *generic instantiation* achieves much the same effect. However, instantiation is a compile-time binding in Ada, and the flexibility of run-time selection of parametric procedures is not available in the language.

4.2.4 Efficiency

Ada contains **pragmas** to influence the default behavior of the compiler. The predefined pragma OPTIMIZE allows a global "hint" to the compiler to be more concerned with either time or space efficiency. The pragma INLINE advises the compiler to eliminate procedure call overhead for specified subprograms. Overall, the pragma mechanism allows a portable, but unreliable, mechanism for influencing efficiency. For the most part, Ada relies on *static type checking* to wring the best efficiency from the compiled code.

Ada allows fine control over main memory representations through *representation specifications*. Objects can be converted between types with differing representations (for example, packed to unpacked). However, Ada provides no direct means to influence secondary storage representation.[11]

Representation clauses specify how the types of the language are to be mapped onto the underlying machine. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware). A type representation clause applies to a type. Such a representation clause applies to all objects that have this type:

- A length clause specifies an amount of storage associated with a type.
- An enumeration representation clause specifies the internal codes for the literals of the enumeration type that is named in the clause.
- An address clause specifies a required address in storage for the entity.

At most one enumeration or record representation clause is allowed for a given type. (On the other hand, more than one length clause can be provided for a given type; moreover, both a length clause and an enumeration or record clause can be provided.) A length clause is the only form of representation clause allowed for a type derived from a parent type that has user-defined derivable subprograms.

An address clause applies to an object. At most one address clause is allowed.

A representation clause and the declaration of the entity to which the clause applies must both occur immediately within the same declarative part or package specification; the declaration must occur before the clause. In the absence of a representation clause for a given declaration, a default representation of this declaration is determined by the implementation.

The interpretation of some of the expressions that appear in representation clauses is implementation dependent (for example, expression specifying addresses. An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware.

Whereas a representation clause is used to impose certain characteristics of the mapping of an entity onto the underlying machine, pragmas can be used to provide an implementation with criteria for its selection of such a mapping. The pragma PACK specifies that storage minimization should be the main criterion when selecting the representation of a record or array type.

Packing means the gaps between the storage areas allocated to consecutive components should be minimized. It need not, however, affect the mapping of each component onto storage. This mapping can itself be influenced by a pragma (or controlled by a representation clause) for the component or component type.

The pragma PACK is the only language-defined representation pragma. Additional representation pragmas may be provided by an implementation. In contrast to representation clauses, a pragma that is not accepted by the implementation is ignored.

---

10. Pragmas can be ignored if not implemented in a particular compiler. Also, compiler implementers are free to introduce new pragmas, providing *more* control.

11. The DEC implementation of Ada provides access to the VMS Record Management Services through Ada's I/O packages. While RMS allows considerable control over record format and storage layout, it is a unique (non-portable) implementation feature.

At most one representation clause is allowed for a given type and a given aspect of its representation. Hence, if an alternative representation is needed, it is necessary to declare a second type, derived from the first, and to specify a different representation for the second type.

4.3 Integration Issues

In this section, we discuss the similarities and differences in the ways Vbase and Ada support the criteria developed in Section 3. Our objective is to identify limitations intrinsic to Ada, and specific areas we will explore during Phase II.

4.3.1 Modularity

Both Vbase and Ada support similar mechanisms for encapsulation and information hiding. Vbase uses **modules**, and Ada uses **packages**. They differ in their semantics for visibility of name scopes, and in their support for inheritance.

4.3.1.1 Inheritance Vbase's type lattice is not a strict tree structure. That is, objects may have more than one parent type. Ada's subtype and derived type mechanisms are strictly hierarchical, although record types may have each component derived from a separate parent type.

4.3.1.2 Visibility Ada has weak support for inheritance, and the compile time static type-checking discipline limits run-time extension of the database. Ada allows object creation at run-time, through the *allocator* mechanism — but not type creation. Vbase allows such dynamic type definition, but the Ada compiler and run-time cannot "see" the type. Therefore, the type checking cannot be performed in Ada (although it will be checked by Vbase at run time). In general, the Ada types will have to follow Ada's semantics for compile-time checking, and Vbase types will have to follow object semantics.

Naming scopes are supported (as named **interfaces**), as are context clauses (**import**). The semantics of interaction between objects in the Database and objects in an Ada program are potentially very difficult. Names in VBase provide bindings between strings (sequences of characters) and objects. Names provide a natural mechanism for referencing objects in a high level language.

Both Ada and Vbase support block structured languages. Each new block defines a new scope for names, which we call a name context. Every name reference is resolved within the current name context (often called a name scope in a programming language such as C, and also referred to as the Environment within VBase). This method of name resolution provides two important advantages:

1. It allows users to restrict the names visible at any given point in the code. Formally, we say a name is visible if it can be resolved to at least one object in the current name context. If the name can be resolved to more than one object, we say it is ambiguous. This restriction of visible names significantly reduces the chance of a name conflict, that is, two objects whose names are indistinguishable within the current name context.
2. It allows the introduction of local names which are not automatically visible outside the defining name context. This again reduces the possibility of name conflicts arising.

A nested name context inherits all of the names of all of the name contexts which contain it. However, if a name is defined inside the nested name context which is identical to a name in an enclosing name context, this is not considered a name conflict. Rather, the name in the nested name context is assumed to be the name desired. In general, when name contexts are nested, names are resolved within the closest enclosing name context. This is the behavior of scopes in programming languages (such as the {..} scope in C, or **begin end** scope in Pascal or Ada). Name contexts in Vbase are defined by **types** and **modules**. Modules are used solely to create different name contexts.

4.3.2 Flexibility

The semantics of Vbase and Ada differ in significant ways. A fundamental difference in the two systems is Ada's insistence on static checking, and on Vbase's support for run-time definitions. (Note, however, that Vbase's type checking is every bit as "strong" as Ada's.)

4.3.2.1 Exceptions Vbase supports parametric exceptions, while Ada's exceptions do not have parameters. The **exception** objects in the database retain their parameters as state information (that is, they are objects, too). Ada exceptions are transient changes to a program's flow of control.

While there is little impact of naming conflicts with (Vbase and Ada) exception handlers, we think there is little advantage to unifying the two mechanisms into a single exception handling mechanism. We will provide the users with the opportunity to handle each exception type separately.

4.3.2.2 Polymorphic Functions Vbase supports "optional" parameters to subprograms, as well as the "default value" mechanism Ada provides. UNIVERS will support the Ada approach, as well as Ada subprogram overloading.

Because Vbase allows run-time type and object definition, it supports run-time type resolution. Essentially, individual methods are dispatched at run-time based on the type of the first parameter supplied to them. The Ada approach to static subprogram overloading resolution is a similar concept. Attempts at using Ada features for run-time resolution (for example, variant records, exhaustive enumeration) are clumsy and inadequate for a truly extensible database. Again, users who take advantage of this feature will have to rely on the more powerful Object Manager mechanism.

4.3.3 Efficiency

In this section, we consider the approaches Ada and Vbase take to providing efficiency in persistent storage. We focus on two areas: expressing representation of objects in storage, and the process structure of the application and database.

4.3.3.1 Representation Just as a type defines the behaviors of its instances, it must also *implement* those behaviors. Implementation is done by translating the behaviors into a set of data structures and transformations on those data structures. Thus, a person might represent the length of time that he has worked at a certain job by scratching marks into the wall of his office: each day, a new mark is made. These marks are NOT the length of time that he has worked there, but they ARE an adequate *representation* for this datum. To be meaningful, however, they must be interpreted.

The task of a type is to define behaviors, choose a representation for those behaviors, and manage the task of translating between the abstract behavior and the representation for that behavior.

4.4 Summary

Table 4-1 summarizes the features provided by Ada and Vbase supporting database extensibility.

TABLE 4-1. Summary of Features

Criterion	Ada Features	Vbase Features
Expressibility	Variables, types, subtypes packages, subprograms and exceptions	Object versions of same (plus iterators) as database entities
Modularity	Separation of specification and body	(Same)
Flexibility	*context clauses* **(with)**	**import** clauses. Run-time *method dispatching*
Efficiency	main memory representation clauses Compile-time optimization	Secondary storage **pragmas** Strong type checking
Persistence	package DIRECT_IO	Automatic storage management

In the next section, we consider how to integrate the two -- providing access to all of Vbase's features to a design application developer working in Ada.

5. Approach

In this section we investigate the integration of the Vbase object oriented database into an Ada environment. This section outlines some potential approaches to providing an Ada-Vbase interface, and describes the goals of our three-phase SBIR project. It summarizes our selected approach by describing how the features discussed in the last section map onto our approach.

It should be possible to integrate external systems into an Ada programming environment, mapping the operations and types of the external system into those of the Ada language. Because the concepts of database schemas and programming languages are similar, we suppose that an external database system can be regarded by the user as an abstract data type: a collection of data together with a set of operations defined on them. The resulting data type definition is called a database schema.[12] The **package** facility of Ada can be used to realize such an abstract data type. The data are described by data types which are described by means of a Data Definition Language (DDL). The operations are defined an a Data Manipulation Language (DML). Our goal is to go beyond providing the usual "embedded language" facilities of a DBMS for Ada.

The Data Definition Language should give the programmer direct support for:

- abstract data types, with separate specification and implementation (**package** and **package body** in Ada).
- a subtype/supertype hierarchy with inheritance of properties and operations down the hierarchy.
- the ability of a type to export access to subsets of the properties and operations it defines through named interfaces; built-in subtype interface which exports implementation information to subtypes which is otherwise unavailable to types using this type.

Data Manipulation Language interface for the data base can be done as Ada **packages**, providing:

- the ability to create persistent objects whose lifetime is longer than that of the process that created them, without having to implement all of the mechanics of copying them out of virtual memory onto files before process termination.
- a more powerful exception handling mechanism
- associative retrieval (queries) built in as instance-identifying expressions.

Normally, the data model within an Ada program library and that of an arbitrary external system are different. The data types and subprograms may not conform to Ada conventions. Therefore integration means the mapping of the types and operations of the external system's to types and subprograms in Ada.

The result of such mapping is called an interface to the external system. The interface will be written in Ada, to support its own extension according to Ada conventions.

The main task of the interface is to pass parameters from an application written in Ada to the database system, and vice versa. Therefore, syntactic (and often semantic) checks are needed. External systems may use a communication area to support this interchange. One objective, then, is to hide this communication area, and leave its management to the same interface mechanism that does the parameter checking. The call to an entry point of the external system itself is done via an external reference.

We can identify the basic goals the interface should meet:

- *expression* of new data structures and data storage layouts
- modular, *incremental* extension of properties and behavior, ideally as refinements of existing descriptions

12. A user is normally not aware of the global schema, but only knows about (that is, has access to) certain parts of the database — the subschema.

- flexible *access* to database extensions (by existing applications and/or utilities) without recompilation
- improved *response time* for gaining access to new database structures

5.1 Vbase Description

As we have seen, Vbase models a broad range of entities as objects with state and dynamic attributes. It allows these attributes to be abstracted by types; types can be abstracted further, by supertypes until the entire application is expressed. But how is this abstract data model realized in terms of an actual software system?

5.1.1 System Elements

While the Vbase data model is powerful and perhaps unfamiliar to many software engineers, the implementation elements are all comfortably familiar. Vbase is implemented with two languages — a procedural language for writing applications programs and a data definition language for defining the type hierarchy. Each of these elements is described briefly below.

The Object Manager is structured as a set of interacting type managers. This approach is in contrast to the more traditional structuring of a large software system into "layers" of functionality. The layering of a software system is intended to enhance its modularity, and provide a distinction between semantics (the interface to a layer) and implementation (the internals of a layer.) While we ensure modularity via the abstraction mechanisms of *classification* and *representation*, it is nonetheless useful to look at the layering which is induced on the types in the Object Manager system. This layering is induced by the interactions between the different type managers, and while it is a loose layering (not strict), it does provide an insight into the overall structure and functioning of the system.

The Object Manager, then, is loosely divided into 5 layers:

1. language layer
2. abstraction layer
3. evolution layer
4. representation layer
5. storage and access layer

This layering is presented in the same sort of ordering as one would normally impose on a traditional software system. That is to say, the topmost layer is presented directly to the user; this layer generally issues calls to the layer below it, and on down the line. Because the actual modularization of the object manager is based on type managers, and not on layers, it is possible for a type manager in one layer to call a type manager in a layer which is not directly below it, or even to call a type manager in a layer above it. The middle three layers implement the features described in the preceding discussion of abstractions. The top layer, Language, interfaces with the user. The bottom layer, Storage and Access, interfaces with the machine.

5.1.2 Language Layer

The language layer of the Object Manager provides the user with access to the facilities of the Object Manager data model. Furthermore, it is the one and only door through which the user can enter. The crucial role of any language is to provide a syntax for specifying the *semantic operations* defined in the base model. A given language may or may not give direct access to all of the underlying features, and it may emphasize or simplify the interface to some features at the expense of others.

5.1.2.1 TDL — Type Definition Language: TDL is the Vbase data description language. TDL statements describe object types and the type hierarchy. TDL is a compiled language.

In most traditional languages, data types are compile time constructs only. Once the program is compiled and variable storage is allocated, their role is ended. In Vbase, on the other hand, type definitions and the type hierarchy are available both at compile time and at runtime. Object type definitions and the type hierarchy play a central role in the system's dynamic behavior.

5.1.2.2 COP — the Vbase Procedural Language: COP (C Object Processor) is a C superset language. It is a true superset. The COP compiler can compile all legal C programs without changes. The most important COP extensions are related to handling objects. These additions have been made organically by extending C constructs to interface directly to Vbase.

COP recognizes calls to object operations as if they were native C function calls. Object types can be used directly in data declarations. Objects in the database can be accessed directly, as if they were program variables without any of the artifacts of file handles and pointers. (However all the standard file calls are still available for conventional operating system files.)

5.1.3 Abstraction Layer

The abstraction layer implements the concepts of **entity, type, property**, and **operation**; and three of the abstraction mechanisms: *aggregation, classification*, and *generalization*.

The **abstraction manager** is the top layer of the Vbase runtime system. It maintains the object type definitions. Type definitions are global across Vbase database and applications programs, similar to a data dictionary in relational systems. This single data description is used by all elements of the system.

Vbase is oriented towards the *abstract data type* paradigm, rather than the *message-sending* paradigm of other object systems.

In Vbase, object behavior is represented through a combination of properties and operations. Properties represent static behavior; objects represent dynamic behavior. Property definition and access are syntactically differentiated from those of operations. This provides a more natural model of object behavior. It also saves the programmer from writing lots of trivial code to get and set the values of properties, as in message-passing systems — in Vbase, these operations are generated by the system, increasing programmer productivity and program quality.

At compile time the type definitions and type hierarchy form a common procedure and type library, and are used much like header files and standard library functions. At runtime the type definitions and type hierarchy are used to dynamically dispatch operations calls and property accesses. Finally, it is used by the underlying storage manager, the Object Kernel, for actual data access.

Thus, the system exhibits the object-oriented flexibility of dynamic binding based on a hierarchy of types. However, it provides performance comparable to a compiled, statically bound system.

5.1.4 Evolution Layer

This layer deals with the recording of history, and with concurrency control and recovery/restart. A single mechanism underlies all three of these functions, which we call the version mechanism. This mechanism allows for both linear evolution and non-linear evolution, as described below.

A *linear history* is one which evolves from a state A to at most one subsequent state B. We call each state of a linearly evolving object a *version* of that object. Each version has all of the static behaviors of the object; only the most recent version (the end version on the linear evolution path) can be mutated, however. Each version of the object has a unique timestamp which identifies it. Any reference to an evolving object must specify what version to use; this can be done by specifying the timestamp, or any other property of the version. If no specification is made, then the latest version is chosen by default.

A *non-linear history* is one which evolves from state A to potentially more than one subsequent state. The multiple successor states to any one starting state are called *alternatives*. Like versions, each alternative has all of the static behaviors of the object, but only alternatives which have no successors (that is, alternatives which are the latest in their sequence) can be mutated.

5.1.5 Representation Layer

The representation layer provides the ability to use instances of one type to represent the abstract behavior of instances of another type. For example, the abstract behavior defined by type STACK might be implemented in terms of an array. The fact that a stack has an array as its internal representation is not visible (nor is it important) to any user of stacks. All that is visible from the outside is the abstract behavior defined by the stack type. In fact, STACK could be re-implemented to use a linked-list representation, completely invisibly to any outside user of the

type. Each type defines not only its abstract behavior (in terms of a behavior specification) but also the implementation of that behavior, in terms of transformations on an internal representation.

The representation layer also enables the use of references to provide an implementation-independent access to objects. A reference to an object acts as a surrogate for the object. As far as a user is concerned, the reference IS the object: it can be operated on by the operations defined by the object's types, etc. But the storage for the object's representation might be located somewhere else, invisible to the user of the object, and not directly accessible. The reference to the object provides the access path to the representation of the object, and this access path is only traversable by the type managers which define the object's behavior. This spec/rep distinction, strengthened by the use of abstract references, ensures that a type is safe from outside damage to its instances. Only the type controls access to its instances' actual storage.

5.1.6 Storage and Access Layer

The **object kernel** is the storage manager. Vbase storage is highly cached for high performance. The object kernel maintains the high-speed data cache and resolves high-level object names to actual storage locations.

Ultimately, the implementation of any behavior must rely on the manipulation of data by code. Both data and code must occupy storage on some machine. The storage and access layer allocates that storage, and provides facilities for managing it. Normally this layer is called by the representation layer, which is concerned with the representation of behaviors. Facilities are provided, however, to allow other layers to interact directly with the storage and access layer, primarily to allow streamlining of the storage subsystem for a particular task.

The database support facilities of Vbase provide automatic management of objects on secondary storage. Vbase also provides:

- sharing of object data among multiple processes/users
- handling large amounts of object data — both number of objects and the total size of object storage space
- maintaining consistent state in spite of system or media failure

5.2 Possible Approaches

This section discusses three approaches towards providing an Ada interface to Vbase. They center around the process architecture and degree of language integration between the programming, data manipulation, and data definition languages.

We are concerned with three primary goals:

- To provide integrated support for persistent objects. The support should be as transparent as possible to users of the system.
- To make the system as type-safe as possible.
- To provide Ada access to the complete Vbase development system — including the database, an interactive object editor for traversal of type and object definitions, a verifier to check physical database consistency, and a debugger for operations/method examination

We can realize these goals through a combination of techniques:

Expression	Extend the Ada language to include Vbase objects — their description and manipulation
Type Safety	All objects in the database will be typed, and we will enforce strong typing both at compile- and at run-time.
Incremental Extension	We will support incremental extensibility, using the inheritance and refinement mechanisms characteristic of object-based systems. This will extend the Ada **package** concept to include the Vbase **module.**
Availability	We are going to provide application-level access to the database schema at both compile- and run-time. Definitions of database types and objects will be immediately available to the defining application (and to others thereafter). We will reflect changes in

Vbase in the Ada Program Library.

Efficient Storage/Access We will provide the means to specify data representations, and separate this from the specification of behavior.

5.2.1 Three Languages

The traditional approach is to use a separate language for data description, manipulation, and general-purpose programming (see Figure 5-1).

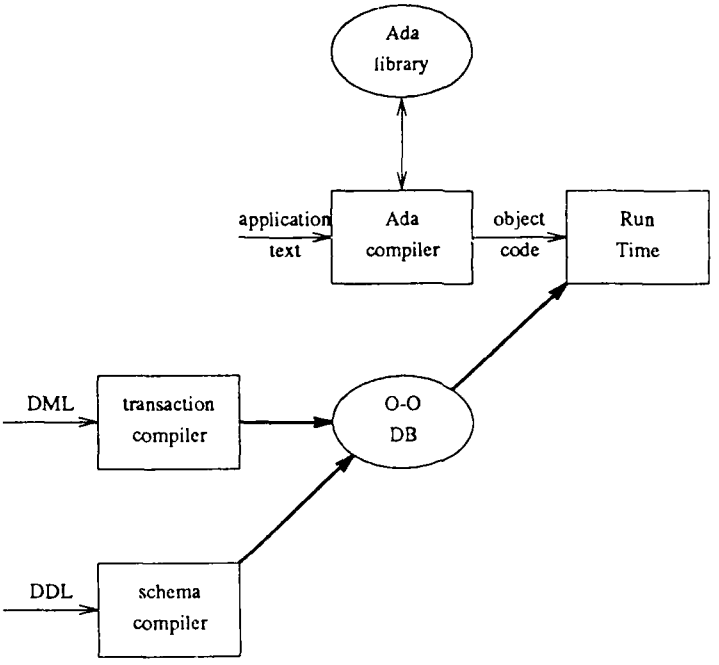


Figure 5-1. Three Languages

5.2.1.1 Features Each language is processed by a separate language compiler, and has independent effects on the database. Typically, the programmer or database administrator first defines new types (schema) and objects in the database via the Data Definition Language. Transactions can be defined via the Data Manipulation Language. Programs written in the programming language may refer to the existing transactions, or invoke database access routines directly.

5.2.1.2 Advantages This approach is the typical, well-proved "embedded language" approach followed by most commercial DBMSs. It's relatively easy to implement, because each language processor is independent and supposedly less complex than the combined system.

5.2.1.3 Disadvantages Using three independent languages is a potential source of errors, as definitions and changes introduced through one mechanism may not agree with the use of those definitions via another.

The key to this approach is to add routines in the program library which allow access to the database. This is often a fixed collection of routines.

It does not support static type checking. Even run-time database references are essentially un-type-checked, unless the database itself contains certain semantic checks.

5.2.2 Two Languages

The first step towards overcoming these difficulties is to integrate some of the database with the programming language. A reasonable first choice is to combine the database operations with the set of subprograms available to an application at run-time. This approach effectively eliminates the need for a separate DML (see Figure 5-2).

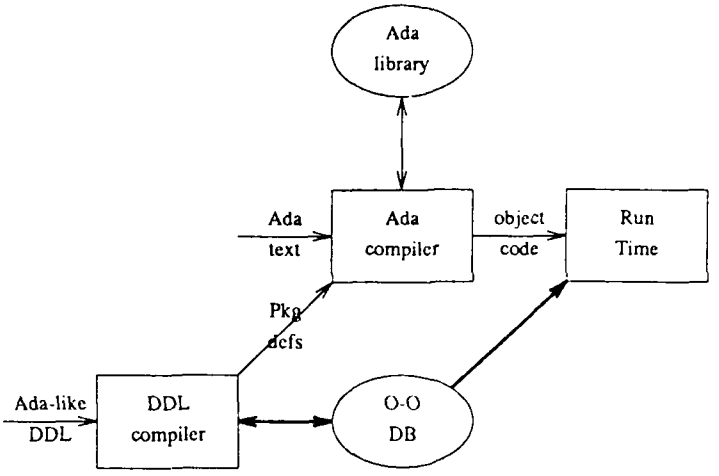


Figure 5-2. Two Languages

5.2.2.1 Features The operators of the DML are mapped to special subprograms like the predefined operators in Ada. The new constructs must conform to the syntax of the programming language, whereas their semantics are those of the corresponding DBMS constructs. The Ada types and subprograms that result represent a *database system interface* which can take the form of an Ada **package.**

5.2.2.2 Advantages By eliminating the DML and its associated transaction compiler, we simplify the total system users must deal with. There is one less language to learn, one less program to run, one less set of options to enable, and so on.

Combining database and Ada operations allows optimizations of those operations through essentially the same mechanisms the compiler already uses.

5.2.2.3 Disadvantages The unified system may be more complex than either component considered separately. For example, query optimizations would be an additional concern for the compiler's code generation phase.

This approach still does not eliminate the DDL — which defines much of the information needed to carry out these optimizations. An additional complexity would be the access by the compiler to the database for this information.

Attempting to provide type checking in the application leads to redeclaring (that is, duplicating) many existing schemas to provide access from Ada. This leads to the usual sorts of maintenance difficulties with duplicate copies of information.

5.2.3 One Language

Since Ada has a rich collection of type definition and manipulation facilities, it seems reasonable to exploit these in an Ada interface to the Object Manager. This "Single Pre-processor" approach would aim to add persistent storage to existing Ada semantics (see Figure 5-3).

This approach would extend the Ada language by integrating the DDL and DML constructs for type declarations and corresponding operators. The DDL constructs are mapped to type declaration constructs, which may introduce type constructors that were not foreseen in the language.

5.2.3.1 Features Another way of looking at this approach is to consider a predefined schema and the operators defined on it as an abstract data type. Abstract data types can be easily realized by Ada **package.** Then the mapping consists of two parts:

1. A mapping that derives the package **type** declarations from the database schema, and
2. A mapping that associates each DML operator with an operation of the package.

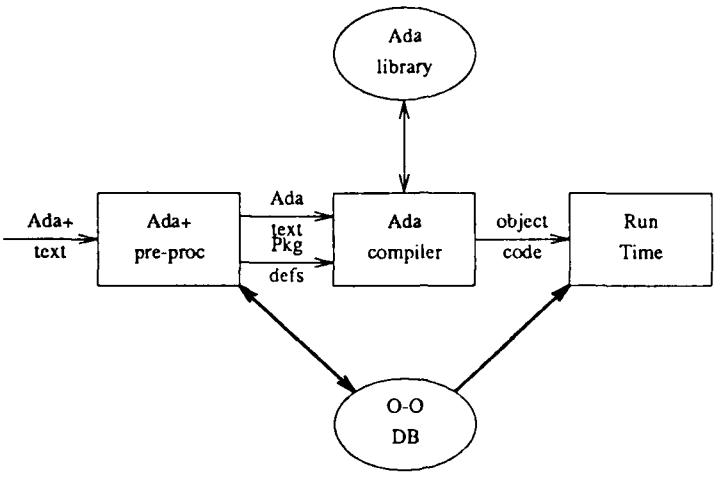


Figure 5-3. One Language

Using the new constructs, the Ada users may declare a database schema in much the same way as they declare **record** or **array** types. In doing so, there must be a reverse mapping which provides a DBMS schema definition (in DDL) and database operators (in DML). The mapping must be expressive enough to provide the equivalent of defining this schema directly in the DDL.

If the schema on the DBMS level and the types on the Ada level have a similar structure, one may even mechanize the mapping: taking a schema as input and producing a set of types in Ada as output. The mapping of operators is a much more complex issue. It involves two steps:

1. Define a mapping that converts the database objects to Ada objects (which may be at least partially *derived* from the data type mapping).
2. Define a mapping that associates a program with each package operator that calls one or more DML operators.

5.2.3.2 Advantages This approach is the best way to integrate the two systems and get complete type checking. The preprocessor picks out Data Definition and Data Manipulation Language statements embedded cleanly in the programming language, and turns these into standard Ada language statements (plus subprogram calls for run-time access). With this integration, we can exploit:

- Strong typing, and compile-time optimization
- Type-specific representations
- Exploiting semantics for performance

It should result in a smaller, more efficient implementation, if Ada semantics overlap with object-oriented semantics. This solution has a benefit from the database point of view: having the schema available on the programming language level will allow type-checking at compile-time.

It also provides a separation of concerns: each subschema to the database must be described separately. Each view is represented by a specific package which restricts the general interface. Each package represents a view for a certain group of users, who may be the only ones authorized to access the package. All such packages form a database view library in the Ada program library. This approach has the added benefit that the structure of the database can be hidden from the users, and that their views are entirely determined by the logical aspects of their applications.

5.2.3.3 Disadvantages The chief disadvantage of this approach (which cannot be avoided in any compile-time solution) is that the Ada programmer cannot create views dynamically at run-time.

With Vbase as the DBMS, there is an additional concern: some common Data Model descriptions may be exceedingly difficult (or even impossible) from Ada. We do not yet know the difficulty of integration (or pre-processing out) for items such as inheritance or iterators.

5.3 Selected Approach

We chose the "Single Pre-processor" approach, adding persistent storage to existing Ada semantics (see Figure 5-3). It should result in a smaller, more efficient implementation, wherever Ada semantics overlap with Object Manager semantics. We ruled out modifications to the Ada compiler allowing it to recognize the extended language directly. This approach carries too great a burden for complying with (indeed, certifying) the ANSI standard semantics of Ada. We also did not choose to merge the Ada program library with Vbase. Although this is the approach taken in one version of the existing Vbase product line,[13] we felt it was too ambitious for Phase II.

No modifications to the Vbase product are required to build the prototype, in which C library functions can be invoked from Ada applications. Thus the Ada interface development can be done independently of the development of the Vbase product. The staff working on this Program will require access to the source code of that interface library only in order to maximize the efficiency of the marriage between the C interface and the Ada interface, and all of the work of optimizing the interaction will be done on the Ada side of the interface, and will be performed under this Program. The Ada interface developed under this Program will be the prototype for an independent layered product compatible with the standard Ontologic Vbase product. That prototype layered product will be delivered to the government as the final deliverable under this Program.

The UNIVERS product will contain a pre-processor and a package-level interface to Vbase in the Ada program library. We will add routines in the program library which allow access to the database. The major benefits of this approach are:

- Integration of the Data Definition, Data Manipulation, and general-purpose Programming languages
- Improved optimization through static type checking
- Ease of extensibility through standard Vbase mechanisms
- Persistence of the symbol table information: types and legality of subprogram parameter profiles
- Access to run-time extensions through Vbase

5.3.1 Pre-Processor

The most complex aspect of UNIVERS is the pre-processor. The purpose of the pre-processor is to recognize the object database extensions added to Ada, and transform the input text into ANSI Ada (with access routines). A side-effect of this transformation process is automatic generation of Ada code, primarily to define new abstract data type packages. The pre-processor includes:

- Access to Vbase for entity descriptions
- Adding type descriptions to the database
- Generating Ada packages to mirror these changes in the Ada program library
- Transforming the input text to ANSI form

The main goal of the pre-processor is to reflect changes in the state of the Vbase database (that is, new type and object definitions) into the Ada program library. This makes the extensions available to the applications written in Ada. Section 6 describes the pre-processor's functions in more detail.

---

13. DBDCOP, the Data Base Dependent C Object Processor. We think this is the next logical step for the UNIVERS product, even though it would require modifications to the host Ada compiler.

5.3.2 Procedural Access to the Database

The main goal of the interface package(s) is to expose Vbase semantics to the design application programmer working in Ada, especially the run-time facilities. The interface package includes:

- Access to the run-time routines of Vbase
- A base context for the automatically-generated Ada packages

Section 7 describes more of the features of this interface.

6. Pre-Processor

This section describes one of the two main components of our approach: the pre-processor. Appendix A presents the grammar of the interface language we've defined. This section highlights some of the features and their support for the extensible database criteria outlined in Section 2.

While Ada has many features which a skilled designer might use for object-oriented programming,[Boo83] it lacks certain features for supporting the dynamic extensibility provided by the Object Manager. We have taken the opportunity to define certain extensions to Ada which facilitate the integration (in the pre-processor) of Ada and Object database features. For the most part, these are ANSI Ada constructs, with the keyword **object** added.

TABLE 6-1. Language Features Summary

Criterion	Language Features
Expressibility	**object, type, variable, constant, properties, operations**
Modularity	**package, procedure, function, iterator**
Flexibility	**method, exception,** *inheritance*
Efficiency	**representation,** *cluster-clause*
Persistence	Vbase

6.1 Declarations

The declarations allow users to describe objects, their types, and their relationships. They also provide ways to define (and limit) the operations allowed for objects, and the structure/representation of the objects.

The pre-processor should take advantage a strict notion of the meaning of subtype as well as modern incremental compilation technology, to eliminate most of this run-time indirection. The application builder should be allowed to "freeze" portions of the type hierarchy once application development has settled down, and compile code for the type managers which is close to the efficiency of typical programming languages.

The pre-processor implements the effect of declarations by modifying the contents of the database. It uses standard Vbase access to define new entities, including types, operations, methods, and exceptions. The pre-processor also generates the Ada packages required to reflect the changed database name space in the Ada program library.

6.2 Names and Expressions

We have kept the semantically distinct name spaces of Ada and Vbase separate by distinguishing them syntactically. The **$** lexical element is not used in Ada, and indicates a database object name. The **$$** symbol has two uses: to anchor a database path name at the root name space in the database; and to the "next" method in a sequence (see the discussion of Statements).

The names in the Ada packages generated by the pre-processor will contain fully qualified names. These names start with the package DATABASE, which must be imported into the context of the application (via a **with** clause).

We have not settled the issues regarding an integration of expression elements in the two name spaces. For now, we will provide little implicit conversions between the Ada and Vbase types. The pre-processor will rely on the normal Ada overloading and subtype semantics to resolve ambiguities.[14]

6.3 Statements

The major operations available to an application will be defined as subprograms in the database interface package (see Section 7). Additional statements recognized by the pre-processor include **iterators** and **exception** handlers.

14. Phase II of the project will investigate this in detail.

The "statement" **$$** indicates the "next" method in sequence, which may be one of:

- A "trigger" in a sequence of triggers.
- A "base method" of the referenced operation.
- A method or trigger on the "supertype" operation.
- A method or trigger on an operation for which the referenced operation is a *refinement*.

All of these statements are transformed by the pre-processor into ANSI Ada, and ultimately realized by the subprograms in the database interface package.

6.4 Implementation

This grammar is not yet complete, to the point where we might implement a working pre-processor. However, it does show the principal constructs of the language. The pre-processor which parses the eventual language will be able to define objects, types, and arbitrary entities into the database. Procedures, functions, iterators, and exceptions can be *objects in the database*, or elements of the Ada compilation library. Their presence in the database directly supports the notion of extensibility we are implementing.

We checked the grammar defined by building a recognizer for the language. We defined a language for the specification (using a modified Backus-Naur Form), and built enough of a recognizer to check the statements for syntactic correctness. This recognizer not only validated the consistency of the grammar, but also gave us some simple syntax checks of the interface itself. We used two language development tools available on UNIX systems, **lex**[Les75] and **yacc**.[Joh75]

6.4.1 Parsing

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user specifies the structures of his input, together with code to be invoked as each such structure is recognized. **yacc** turns such a specification into a subroutine that handles the input process.

The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules. The theory underlying **yacc** has been described elsewhere.[Aho74, Aho75a, Aho77] While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

6.4.2 Lexical Analysis

lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to **lex**. The **lex** written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The **lex** source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by **lex**, the corresponding fragment is executed.

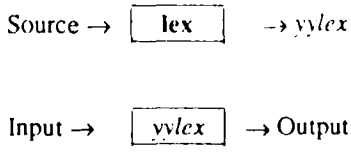


Figure 6-1. An overview of lex

lex turns the user's expressions and actions (*source*) into the host general-purpose language (see Figure 6-1); the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input*) and

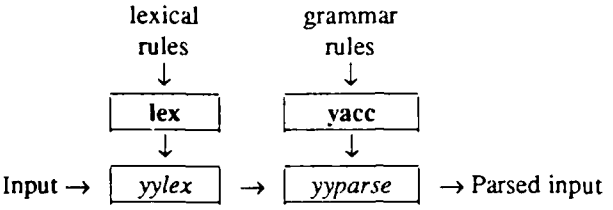


Figure 6-2. lex with yacc

perform the specified actions for each expression as it is detected.

lex programs recognize only regular expressions; yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and yacc is often appropriate. When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 6-2. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**.

lex generates a deterministic finite automaton from the regular expressions in the source.[Aho75b] The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number of **lex** rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by **lex**.

6.4.3 Recognizer

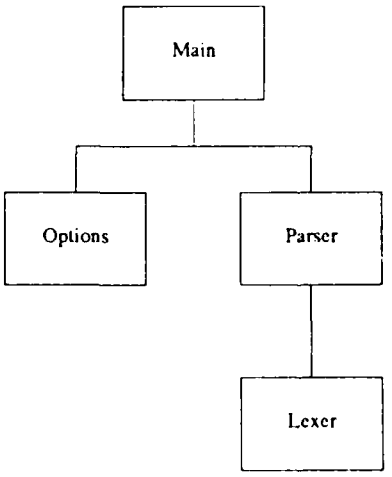


Figure 6-3. Recognizer Software Hierarchy

Figure 6-3 illustrates the software hierarchy. The resulting recognizer is a grammar/syntax analysis tool for the UNIVERS language, and was produced from a public-domain **yacc** specification developed for the ANSI Ada standard. The software is included as Appendix E.

This tool will do little than check the syntax of a UNIVERS program. We used it to check the data model described in Section 8.

7. Interface

This section describes the other main piece of our system, the interface between Ada and the Vbase Object Oriented Data Base. It explains in more detail the interaction between the pre-processor, the Ada program library, the run-time library linked with the application, and Vbase.

The Ada program library interface has to perform three functions:

- Provide access to *existing* Vbase Abstraction Manager and Representation layer function calls.
- Implement the Ada packages automatically generated by the pre-processor.
- Generate the application code to run on top of the first two items.

We will provide these facilities as Ada packages, in the same way the basic run-time environment is described in package SYSTEM. The basic Vbase services will be described in package DATABASE. Packages generated by the pre-processor will include this package in their context.

7.1 Library of Facilities

Package DATABASE provides the definition of types, objects, and operations necessary to use Vbase in its most elementary form. It constitutes the run-time library for an application. Theoretically, careful Ada programmers could use this package directly — without invoking the pre-processor. However, this practice denies them most of the power of Vbase's abstraction and extensibility mechanisms.

```
package DATABASE is
   declarations of types and subtypes
   declarations of objects
   declarations of subprograms
   declarations of exceptions
private
   representations of any private types
   pragmas for any external subprograms
end DATABASE;
```

7.2 Run-Time System

A feature which can reduce storage requirements and improve performance for systems involving small objects is the support for type-specific value inheritance down the **a-part-of** hierarchy. If values for properties can be inherited by an object at a lower level in the hierarchy, then they need not be physically stored as part of the representation of each and every lower level object instance. A performance-related side effect of this notion of property value inheritance down an **a-part-of** hierarchy is that it is straightforward to refine the *get_property_value* and *set_property_value* operations on the types in the **a-part-of** hierarchy to cache current property values in a 'state machine' from which they can be very rapidly retrieved without even going to the underlying database.

The UNIVERS product should provide the type safety and efficient performance of Vbase whenever possible:

- Type specific representations.
- Exploiting semantics for performance: clustering, and anticipatory pre-fetch.

Vbase does not force all objects to be implemented on top of a single low-level primitive (e.g., record or tuple). Instead, it is be possible to define a custom implementation for each distinct type.

Many of the early object systems faulted individual objects into memory. That works well for large objects, but breaks down for small objects. When objects are small (as in polygons for a VLSI mask), many hundreds or thousands of them may be transferred in as a unit. The Vbase system contains storage pragmas which allow application builders sophisticated control over how it clusters objects into segments.

Most traditional systems have the database running in one process and the applications in separate processes. The application must send an Inter Process Communication call to the Data Base Management System to request data,

which can take from 7-15 milliseconds: nearly half of the time required by a disk access. At 10 milliseconds per call, even if the Data Base Management System itself was infinitely fast and took no time to process the case, that would be a maximum of 100 calls per second.

Most engineering applications operate with their design data mapped directly into their address space. Vbase maps its own code as well as large segments of Object Memory directly into the address space of the process using it. This means that the call overhead is analogous to that of a subroutine call rather than an Inter Process Communication message — 50 microseconds rather than 10 milliseconds. The theoretical maximum number goes up from 100 to 20,000 calls per second.

All of these facilities of Vbase should be expressible in the language recognized by the pre-processor. However, in early implementations of the UNIVERS product, these facilities will not be available at the language level. Instead, the programmer will have to use direct access to Vbase facilities. This access will be provided to the Ada applications via use of **pragma** EXTERNAL — thereby exploiting both the ususal flexibility of Ada to cope with foreign code, and the investment in the working Vbase system.

8. Examples

This section provides examples of Object Oriented Data Base extensibility. It uses the grammar we've defined to express elements of the Data Model which should be accessible through the interface we've described. Finally, it demonstrates the dtabase extensibility available using Vbase — by using the Data Model to extend itself.

8.1 Data Model

A data model is the set of abstractions underlying a particular programming style or programming environment. For example, the data model of the popular languages like Pascal or C includes constructs like the type definition, the binding of variables to type definitions, the languages' block structure, functions and procedures as objects with their own private state and procedure information, etc. Similarly, records, keys and files are the abstractions at the root of relational databases.

Data model abstractions form a bridge between the problem and its solution. Thus they must be good implementation constructs and model the application well at the same time. A model invariably requires extra work to implement if it lacks power to express either reasonable implementation constructs or relationships inherent in the problem. For instance, consider the file/record/field data storage model that evolved from the Hollerith punched card. It is an easy model to implement, but it is not very expressive of the relationships intrinsic in the data. Therefore systems that use this model or its newer derivatives must express data relationships outside the model, in the applications programs. This invariably results in duplicate code for expressing similar relationships in different contexts and leads in turn to more work, more opportunity for errors and more effort for maintenance and enhancements.

We will use the language defined in the Section 6 to describe the interface precisely, by enumerating database objects and defining them in terms of the supplied formal language. These objects are the "kernel" of the database. Everything in the database is described in terms of these objects. The interacting mechanisms of *inheritance* and *refinement*, together with these definitions, provide extensibility in the database.

8.1.1 Types

The contents of the Vbase database are defined by its types and their properties and operations. Every object in Vbase is ultimately a type of Entity, which means that every object has at least the behavior of an Entity. Create, Delete, Get Value, and Set Value are operations that can be performed on all objects.

Entities in Vbase are commonly decomposed into types and instances of types. Instances of type Type also define the behavior of their instances, so that they are types as well as instances.

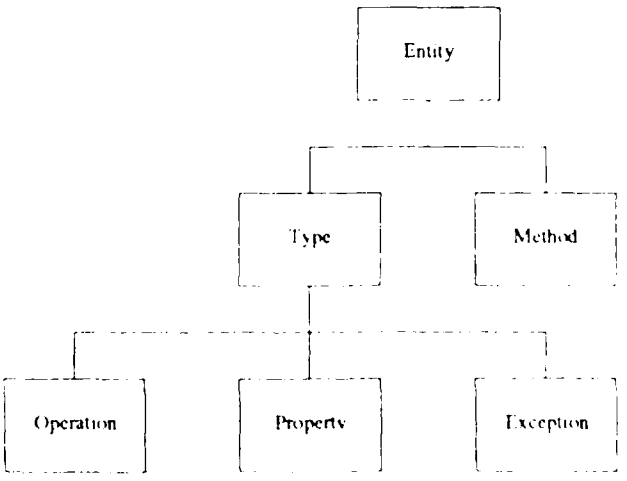


Figure 8-1. Summary Type Hierarchy

Figure 8-1 shows the most integral types of the Vbase data model, and their relations to each other. When users add

new types, they are actually extending the Vbase data model, while at the same time keeping the full support of the database. The types of the Vbase data model themselves, however, cannot be modified by the user, since they define the fundamental behavior of the Vbase system.

The type Type defines the behavior of its instances. Even Entity and Type have type definitions, which makes every part of Vbase extensible, and makes upward compatibility easier to provide. The type Type in Vbase corresponds to *class* in Smalltalk-80, and to *relation* in relational databases. A type defines the properties and operations, inherited properties and operations, and implementations of an object.

The following subsections describe the Data Model. The actual listings for them are included as Appendix F, expressed in the extended Ada language recognized by the UNIVERS pre-processor.

8.1.2 Entity

All entities, including type Entity itself, are direct instances of some type. This behavior is specified by type Entity itself (see the directType instance property). The direct type of type Entity is called MasterType. The behavior of master types is described in the section on MasterType.

All things in the database are entities, and all entities acquire their behavior from the types of which they are instances. These statements apply equally well to types themselves. In order to avoid self-definitions, or infinite numbers of types, the behavior for type Entity is acquired by its being a subtype of itself. This information is of no particular value to the application writer, but is visible. If you ask for the supertype of type Entity, you'll get back a set containing type Entity.

While all objects in the database are ultimately instances of type Entity, there are NO objects which are Entity's and nothing else (such an object would be rather amorphous). The fact that type Entity's instantiable property is FALSE means that it is NOT possible to create a raw Entity.

Every entity is a direct instance of some type. This type may be accessed via the entity's directType property or via the typeOf operation described below. Either method of access produces the same result, and one is in fact defined in terms of the other. It is a matter of programming convenience and taste as to which should be used.

It is always possible to determine whether two entities are identically the same. This operation returns TRUE if its arguments are in fact the same Entity, and FALSE otherwise. The definition of *identically the same* is very strict. Exact structural equivalence, for example, does NOT imply equal identity. Neither is the floating point number 3.0 identically equal to the integer number 3 (they are numerically equal, but behaviorally different). The meaning of *equal*, is that the two entities have exactly the same behavior, and that if either is ever modified in any way, then that modification is visible in the other.

The Entity$delete operation attempts to negate the existence of its argument. If successful, the entity deleted will become inaccessible, in the sense that no further operations or property accesses may be performed on the entity. The delete operation may be unsuccessful for a variety of reasons, of which three are common. The first is that the entity is a universal (it is not possible to delete the number 3 for example) which results in a CannotDeleteUniversal exception. The second is that the entity is the value of a required property of another entity, such that deleting the entity would violate the integrity of the other. This results in a DeletionConstraintViolation exception. The final typical exception occurs when low-level resources required to effect the delete are temporarily unavailable: for example, if the media on which the entity resides is "down". All of these exceptions are subtypes of the CannotDelete exception.

There are several important things to know about deleting an entity. The first is that references to an entity may exist even after the entity is deleted. If, for example, Pipe A has a property which connects it to Pipe B and Pipe B is subsequently deleted, pipe A will continue to refer to pipe B unless explicit action is taken. This is called the "dangling reference" problem. It is the responsibility of the deletor to ensure that properties refering to the deleted entity are also deleted. This is normally done by refining the Entity$delete operation.

A second thing to know about Entity$delete is that it is *shallow*. Deleting an entity will NOT automatically delete all of its constituent parts. This can be effected, if desired, by refining the Entity$delete operation.

It is always meaningful to ask whether a given entity is an instance of a given type. Entity$hasType will return TRUE if the relation exists, and FALSE otherwise. If an entity has a given type, then it is guaranteed to exhibit all

the behavior defined by that type and each of that type's supertypes. In other words, the entity will have all properties (although not necessarily property values), defined by the type, and all operations defined by the type can be meaningfully applied to the entity.

Because the language supports strict type checking, it is never necessary to ask whether an a variable has its declared type, or any of its declared type's supertypes - the answer is always TRUE. The two times it makes good sense to ask **hasType (e, t)**, are when either the declared type of **e** was a union of several types, or when **t** is a subtype of the declared type of **e**.

Every entity is an instance of one or more types, and acquires all of its behavior from its types.[1] The Entity$typeOf operation always returns the one direct type of an entity. An application might use the operation for the same reasons as using hasType.

The Entity$getTypes iterator successively yields all of the types of which an entity is an instance. This information can be used to determine the complete behavior of the entity.

8.1.3 Type

Type TYPE is an instance of itself. In other words, it is a type; just like type PERSON or type PIPE are types. The purpose of types is to define behavior for their instances. Type PERSON might define a Person$marry operation, and type PIPE might define a Pipe$diameter property. Type TYPE defines behavior for its instances too. This behavior is the ability to define behavior. Type TYPE defines the ability for ITS instances (types) to define operations and properties for THEIR instances. It is because type PERSON is an instance of type TYPE, that type PERSON can define the marry operation. In like fashion, it is because type TYPE is an instance of itself that type TYPE can define the ability to define.

Type TYPE is a subtype of type Entity, from which it inherits behavior. The behavior which Entity defines are the create, delete, equal operations, and the typeOf property. The fact that type TYPE is a subtype of type Entity means that it is possible to create new types, delete old types, test two types for equality, and for types to themselves have a type. Furthermore, we don't have to repeat these capabilities in the definition of TYPE; they are inherited from Entity. The ability for type TYPE to be an instance of itself (or anything at all for that matter) is derived from the fact that it is a subtype of type Entity. It is interesting to note that type TYPE defines the Type$supertypes property. The reason that it has a supertype itself is that, as stated above, type TYPE is an instance of itself.

Any entity may be named, but types may additionally "know" their name. The distinction between an entity having a name (being named) and knowing its name is subtle and not of great concern to the application writer. The value of the name property is not a string of characters, but rather a symbol. A symbol contains a string of characters, and a reference to the entity named by that string. Thus the Type$name property functions somewhat like an inverse property.

Every type must be a subtype of some other types, from which it inherits behavior. The value of the Type$supertypes property is a set of types rather than a single type. If one of the supertypes of type A is type B, then type A is automatically one of the subtypes of type B. This is to say the the supertypes and subtypes properties are inverses of each other.

For the most part, it will be rare for the application programmer to be concerned with Type$supertypes and Type$subtypes as properties of type TYPE. The main motivation for dealing with them (again, as properties) is for dynamic (ie runtime) type creation. When defining a type, however, the application writer WILL be concerned with the supertype(s) of the type being defined, for it is via the supertypes that behavior is inherited. The user will probably never explicitly declare a type's subtypes as these are derived by the system.

1 In the current release, an entity is restricted to being a **direct** instance of only a single type, although it is also an **indirect** instance of all of its direct type's supertypes. In plainer English, a person cannot currently be an instance of both type STUDENT and type PROFESSOR

The operations which a type defines for its instances are recorded in the type's Type$defops property. The value of this property is the set of all such operations defined. Each such operation is related back to the type which defined it by an inverse property Operation$basetype.

The existence of the Type$defops property is normally of little concern to the application-writer/type-programmer, because the language supplies a clearer syntax for defining instance operations — the **operations** construct. The main use of the defops property, then, is for dynamic type creation and modification. The properties which a type defines for its instances are recorded in the type's Type$defprops property. The value of this property is the set of all such properties defined. Each such property is related back to the type which defined it by an inverse property Property$basetype. The existence of the Type$defprops property is normally of little concern to the application-writer/type-programmer, because the language supplies a clearer syntax for defining instance properties -- the **properties** construct. The main use of the defprops property, then, is for dynamic type creation and modification.

Each type has control over whether or not it is possible or meaningful to have direct instances of that type (ie instances of the type which are NOT also instances of some more specific subtype). If a type IS directly instantiable, then it MUST supply concrete implementations for each of the operations it defines. If a type is NOT directly instantiable, it may supply all or none of the methods for the operations it defines (counting on subtypes to supply missing methods). By default (ie if the programmer says nothing to the contrary), new types ARE directly instantiable. The Type$instantiable property is initialized at type creation time. Its value may be subsequently retrieved (although an application would not have much reason for doing so), but may not be modified. Needless to say, if a type is uninstantiable, the compiler will ensure that it is not possible to create a direct instance of it. Nevertheless, the type programmer may still wish to provide a create method or create triggers which will get invoked when an instance of a subtype of the non-instantiable type is created.

All types have an associated class of their instances. In other words, it is always possible to iterate over the instances of a given type. Because some classes are very large or infinite (and the iteration may not end in reasonable time), we differentiate between explicit and implicit classes. An explicit class caches its instances so as to be able to return them quickly. An implicit class aggregates its instances by a predicate which selects them from among all entities. By default, a type's class is implicit.

8.1.4 Property

Type PROPERTY is a subtype of type TYPE. This means that instances of PROPERTY will also be instances of TYPE — that is, they will be types which can have instances of their own. When type PERSON defines the Person$address property, this means that an individual person (say, Craig) can have an individual address property. Craig's address property would be an instance of the Person$address property. The rationale behind this is twofold. First, it allows individual properties to be denotable entities which can be used as any other entity, ie. passed as operation arguments, assigned as the values of other properties, etc. Second, it allows a PROPERTY to define new behavior for its instances. This means that individual properties can themselves have properties, and that they can have property-specific operations defined on them. Information which might be specific to Person$address is whether the address is a mailing or home address, how long the person has "been at" that address, whether the person owns or rents that address, etc.

Every PROPERTY is associated with the TYPE which defined it. This information is stored in the PROPERTY's basetype property. It is the inverse of the type$defprops property. The system uses the PROPERTY$basetype property in the implementation of property manipulation. It is of interest to the application writer only for dynamic (run-time) type creation or modification.

A property relates a subject to a value. For reasons of type safety, it is necessary to make some statement about the class of permissible property values. The PROPERTY$vspec property records this statement in the form of a valuespec. Most valuespecs are simply a type. The Person$mother property might have a vspec of FEMALE - which means that the value of any mother property must be a FEMALE. Sometimes, it is useful to allow a union of types as a vspec, or even a parameterization.

The value of a PROPERTY is manipulated by *get* and *set* operations, which respectively retrieve and update the property's value. By default, the system automatically generates these operations, so that it is sufficient for the type programmer to define properties without having to write any code. In some cases, however, the type programmer

may wish to override these defaults with her/his own methods. This is typically the case if property values are derived, or if a special representation is used, or if certain constraint-checking or side-effects must occur upon access. For this reason, every property has *get* and *set* properties whose value is an operation type. The particular operations used must be specified at the time of the PROPERTY's creation, and can never be subsequently changed.

PROPERTYs are either optional or required. If required, any instance of the property's basetype must always have such a property. This means (among other things) that any attempt to create an instance of some type must supply, as arguments, values for all of its required properties. The system uses optionalProp to validate (among other things) creation attempts. The application writer typically specifies the value of this property implicitly via the **optional** clause. The main reason to know about **optionalProp** explicitly is for dynamic type creation.

It is possible to specify a default value for optional properties (the information, if supplied, is ignored for required properties). This default value must, of course, meet the vspec (i.e., must be a legitimate value). It is currently used only at instance creation time to supply an initial value. If an explicit property value is supplied at creation time, the default value is not used. Typically, the application writer specifies a default value implicitly via the := construct. The main reason to know about it explicitly is for dynamic type creation.

PROPERTIES can have inverses, examples being parent-child, employer-employee, etc. The system guarantees that a change to one part of an inverse will automatically cause a consistent change to the other part. If a person changes her/his employer, then the system ensures that a) the new employer refers back to the person as its employee, and b) the old employer no longer refers to the person as its employee. It is important to note that required inverse properties cannot be changed. It is therefore advisable to specify inverse properties as being optional.

When a PROPERTY can be multi-valued, the meaning of an inverse may apply either to the aggregate of values or to each individual value. The latter case is indicated by setting the PROPERTY$distributedProp to TRUE.

The three options above relate to the ability to embellish previously defined properties. A PROPERTY may either allow or prohibit subsequent refinements. The choice generally represents a trade-off between ease of modification (refinable) and speed of execution (unrefinable). In general, it is advisable to allow properties to be refined during development, and to "freeze" the schema (for performance improvement) when in "production".

Given that a PROPERTY is refinable, it is still possible to limit the ways in which it can be refined. A FALSE value for the constrainableProp indicates that a property cannot be refined by constraining its valuespec (restriction on legal values) to be a sub-valuespec of the originally defined valuespec. Allowing constrained valuespecs in property refinements is both a powerful and a dangerous capability. It is dangerous in that code which works for a given type might stop working when a subtype (which constrains the valuespec) is added to the schema.

A property which is a refinement of another property refers to the latter via the PROPERTY$refinesProp property. This property is used by the system to implement property refinement, and is of interest to the application writer principally for dynamic (run-time) type creation.

8.1.5 Operation

Type OPERATION is a subtype of type TYPE. This means that instances of OPERATION are also types, and can have instances of their own. The instances of a particular Operation are viewed as running code - a stack frame in progress. Thus, Person$marry might be an instance of OPERATION. Its instances would be invocations of the marry operation, i.e., weddings in progress.

OPERATIONs are implemented by methods. Methods are code written in the language. The arguments, returns, and exceptions of a method must match those formally specified by the operation it implements. An operation refers to its method via the OPERATION$baseMethod property; and inversely, the method refers to the operation(s) it implements via the Method$implements property.[2]

Every OPERATION type specifies its formal interface in terms of its arguments, its returns, and its exceptions. The argument specification limits the valid inputs to the OPERATION type. The return specification describes the outputs of the OPERATION type upon normal termination. The exception specification details the particular kinds of exceptional situations which may arise during processing, and the information returned in these cases.

A trigger is a piece of code which is executed in response to a given event. The invocation of an OPERATION type is one of the kinds of events which an application writer may wish to monitor. By attaching triggers to operations, the application writer may perform a variety of functions, including performance monitoring, constraint enforcement, value derivation, and implementing certain kinds of inheritance not directly supported by the model.

A trigger envelops the OPERATION to which it is attached. A portion of the code is executed first, then control is passed to the operation proper, and then the remainder of the trigger code is executed. In this way, the trigger can "see" the state of the world both immediately before and immediately after the operation is invoked (NB: state - local variable values - is preserved across the operation invocation). How much code is executed before or after the operation invocation is controlled by the trigger writer by appropriately placing the $$ statement (which invokes the enveloped operation).

Multiple triggers may be attached to a single OPERATION. Each new trigger envelops all triggers previously defined and the base OPERATION. In otherwords, triggers nest in definition order.

Some operations are defined by a type, for application upon its instances. Other operations are free-standing. The former are called **type-ops**, and are somewhat like messages in a message-passing system. The latter are called **free-ops**, and are somewhat like procedural-abstractions. Choosing between the two is largely a matter of taste. When an operation affects the state of only one of its arguments, it is generally described as a type-op, defined by the type of the argument whose state it modifies. When an operation is not particularly well-bound to any of its arguments, it tends to be described as a free-op.

While in most cases operations can be modeled either as type-ops or free-ops, there is a functional difference between the two. Type-ops may have either subtypes or refinements, but free-ops may only have subtypes - no refinements. The important point about type-ops and refinements is that dispatching occurs upon operation invocation. The effect of dispatching is to ensure that an appropriate piece of code is run - based on the type of the dispatch argument. This means that type-ops will avoid some run time type checks, and are on the whole slightly more resilient to change than free-ops.

The following properties have to do with refinements, and are therefore only meaningful if the OPERATION$basetype property is non-null (ie if the operation is a type-op).

An operation refinement is an embellishment upon a previous definition. An embellishment can take many forms, but can NEVER contradict the previous definition. The most common form of refinement is reimplementation of the same operation specification. This is called method replacement. Other legitimate refinements include:

o Adding triggers to the defined operation
o Adding new optional arguments
o Making previously required arguments optional with supplied defaults
o Changing the declared type of a dispatch (1st argument) to a subtype
o Changing the declared type of a non-dispatch argument to a supertype
o Changing the declared type of the return value to a subtype
o Eliminating previously defined exceptions from the specification
o Adding new exceptions, which are subtypes of previously defined exceptions

It is up to the type programmer to decide whether or not an OPERATION type may be refined. Because of the strict stance on permissable refinements, no harm will ever come from allowing refinements; however, certain compiler optimizations are possible or easier if refinements are forbidden. The set of all refinements of an operation

2. In the current release, an OPERATION must have one and only one implementation; however, a method may implement more than one operation. In future releases alternate implementations (that is, multiple methods per operation) may be allowed.

is stored as the value of its OPERATION$refinements property. The OPERATION$refinesOp property is its inverse, and denotes the operation which is refined.

The overall order of method combination (ie execution) in an environment with both triggers and refinements is somewhat complicated, but important for the application writer to know. First, the most specific refinement is located - this is called method dispatching. Next, triggers of that refinement are executed in trigger-definition-order. Next the base method of the refinement is executed.

8.2 Extensibility Using Vbase

The user of the Object Manager adds new behaviors to the system by creating new types. Our fundamental modeling construct is the ENTITY. The task of creating an application is largely a task of creating a set of types which model the behaviors in the application's universe. Objects can also model software constructs. For instance the *Sort* object defines a particular collating sequence and all the ordering operations in one object.

Objects are defined by type definitions, much like C or Pascal variables are defined by typedef or type declarations. Type definitions specify both the object's data structure and its operations and serve as a template for the object. The object, then, can be thought of as a specific *instance* of its type.

For example if the machine part object in the CAD example is overstressed, it will fail just like the real part will. In other words, if we call the operation StressCalc with too high a value for load, it will return the value fail. The details of the calculation and the specific stress limits are related to the object — to its materials, and geometry. So, in an object system they are bound to the object, not the program. The application program can handle all objects in the same way, regardless of the algorithms used to calculate stresses and determine failures.

8.2.1 Types

The specification of a type consists of three parts: a set of operation types, a set of property types, and an invariant. The type which has this behavior spec will be called the *base type*. The meaning of this spec is, roughly, that instances of the base type can be operated on by instances of the operation types, can have instances of the property types, and will always satisfy the invariant.

For example, type CAR defines (in its behavior spec) the operation Refuel. This means that an instance of type CAR can have this operation applied to it; that is, cars can be refueled. This operation cannot be applied indiscriminately to all instances of CAR, however. The operation type itself has a specification, which describes under what conditions it can be invoked, what the results of that invocation will be, etc. The rules for invoking an operation will be discussed under the heading of Operations, but for now we will make the generalization (not always true) that an operation can be applied to any instance of the type which defines the operation.

Type CAR might also define the property Enginesize. Instances of CAR can therefore have an Enginesize property, whose value is a volume expressed in cubic inches. Again, the specifics of when a car can have such a property, what its value will be, etc., are defined by the spec of the property type. But for now we will assume that all instances of a type can have all properties defined by the type.

The invariant defined in a type's behavior spec is a predicate, which is asserted to be true of instances of the type. This invariant is not usually tested (although it can be, for debugging purposes): the type simply declares the invariant to be true, and this fact is available both to users of the type, and to the compiler. The invariant is defined in the context of an instance of the type, and can reference properties of the instance, etc.

Type definitions are divided into two sections — a publicly-visible section defining what the type *does* and a private, hidden section defining how it is *implemented*.

The public section defines the publicly accessible properties and operations. These, in effect, comprise the object's "guaranteed" external behavior. Changes to these will affect applications programs using the object.

The **private** section defines the object type's implementation and is generally hidden from external view. The implementation section can include special properties required by the object itself. It can also define the data storage structure (called the *representation*) that is used. The representation can be chosen for high performance or for greatest data compression, for instance. In fact, since all property access is via operations, an object's properties could be derived and not stored at all.

In Vbase, operations on a type are specified in the type definition. The operation Part_display of the Part type

```
object type Part is
  ...
  operations are
    Part_display (p: Part)
    method (shuttle_part_display);
  end operations;
end Part;
```

is a simple operation specification. The name of the operation, Part_display, is followed by the argument list enclosed in parentheses. In this case only one argument is passed, identified by the label p. To the right of the colon following p is the valuespec of the p argument, in this case Part. All operations must have at least one argument, which is the dispatch argument.

The first argument supplied in an operation is the dispatched argument, which is always of the type for which the operation is defined. Dispatching guarantees that the proper method for the operation is used. For instance, suppose a subtype of Part, for example, Structural_Part, refines the operation Part_display. Whenever Part_display is called with a specific part the appropriate method will be used automatically. In other words, Structural_Part's method will be used if the particular part on which the operation is invoked is also of type Structural_Part. Operations are restricted to a single dispatch argument, but other arguments can be passed.

An operation performs an action on one or more arguments, and terminates returning zero or one result. Communication between the operation invoker and the operation occurs through the arguments and results. The operation Part_display could change p, since p is passed by reference. References are abstract pointers to entities. Entities are never directly manipulated, but are accessed through references. All arguments and return values are actually passed as references to an Entity, rather than as the entity itself.

The **return** statement constrains the operation to return an object of the type specified. In the example below, the operation Part_display normally returns an object of type Part.

```
Part_display (p: Part )
method (shuttle_part_display)
return (Part);
```

Only one type may be specified in a return statement.

The **method** statement declares the name of the subroutine which implements this operation. The method for Part_display is named shuttle_part_display. An operation can have only one method, but the same method can be used by several operations.

An operation can terminate by returning the normal result, or, if that is not possible, by returning some notice of an error. This is called raising an **exception.** In order to be able to raise an exception, the operation specification must include the raises statement as shown below:

```
Part_display (p:Part )
  method (shuttle_part_display)
  raises (NoImage);
```

The operation Part_display can only raise the NoImage exception (or one of NoImage's subtypes), because that is the only exception specification defined.

8.2.1.1 Properties A *property* is an Entity which relates two other entities. One of the entities is generally isolated as the *subject*, and the other is the *value* of the property. (For example, "the sky is blue" is a statement which asserts that the sky has a color property, with the value 'blue'. 'Sky' is the subject; 'color' is the property; and 'blue' is the value.) Properties are the basic information-carrying constructs in the Object Manager, and model the *state* of an entity.

Properties model an entity's *attributes* and its *relationships* with other entities; they represent the state of an entity. They are defined in terms of a set of operations, which access that state. The behavior of a property is specified by a

property spec, which describes how the property-access operations interact. A property can be thought of as a directional link between two entities, which are called the *subject* and *value* of the property.

The object's properties can be simple or arbitrarily complex. They can range from a single basic value like an integer to a complicated structure consisting of string values, arrays, sets or, in fact, any other objects in the system.

Properties in Vbase can be regarded as attributes or record fields like traditional database systems, but that ignores some of the benefits of the Vbase design. Objects have a state as they do in the real world, and their qualities change as their behavior changes. A property is actually a special form of operation that assumes physical storage of the result. This opens a wide range for property behavior--in fact, the whole range of programmable behavior that is available for operations. Properties are a shorthand for a common database requirement, that is, storage and retrieval of named values of objects.

8.2.1.2 Operations *Operations* are the active elements in the Object Manager: an operation can *create*, *access*, or *mutate* objects. Nothing can be done to any object other than to apply operations to it. Operations are essentially functions (or procedures) and can be called from applications programs. When they are called, they are passed arguments and may return an object, like conventional functions.

Operations define the interface between the object and the external world. Thus, at a minimum, all objects have *get* and *set* operations for reading and writing object properties. However, they may have any number of additional operations. In fact, the operations may be essentially equivalent to library functions and use the object's properties for storage of their local state.

Operations are procedural abstractions that are implemented by writing code. All interactions with the Object Manager are performed by the execution of operations. An operation has associated with it a piece of code, called a method. Operations can be invoked, where invocations consist of instantiating the operation type; initializing its state, including its argument list and local variables; and executing the code-block, or method, associated with the operation. The operation instance is used in the same way that a piece of code accesses the state stored on the "stack" in a traditional stack-oriented language.

8.2.1.3 Refining Operations An operation refinement can change the implementation of the operation, or the specification of the operation, or both. Changing the operation implementation means adding new methods. Changing the operation specification means changing the argument, return, or exception specifications.

A type's operation can only refine its supertype's operations. Each operation refinement contributes behavior, that is, each defines a set of local variables, and can place additional constraints on the invocation spec and termination spec. Each operation refinement can also contribute code pieces, or methods. When the operation is invoked, an operation instance is created, and the methods are combined. The method associated with the lowest level type is executed first, and, if the $$ call is part of the method code, it calls the supertype operation, on up the line to the topmost supertype.

When an attempt is made to invoke an operation, the type of the first argument of the argument list is compared with the type of the operation and all its refinements. The lowest-level refinement whose type is allowed is chosen for instantiation. This is called the dispatching process, and the first argument is the dispatch argument. Dispatching is the first action of the Object Manager when an operation is invoked. The second action is the actual execution of the operation, and the third is the return of the operation to its invoker.

A refining operation must refine the dispatch argument to match the subtype. For example,

```
object type Structural_Part is
  ...
  operations are
    refines Part_display (p : Structural_Part)
        method (structural_Part_display);
  end operations;
end Structural_Part;
```

specifies a different method for the Part_display operation, which only operates on structural parts rather than all parts.

No other changes to the argument list can be made (such as adding or dropping arguments).

The returns statement can be changed to specify the subtype value specification. For instance,

```
Part_display (p : Part)
  return (Part);
```

can be refined to

```
Part_display (p: Structural_Part)
  return (Structural_Part);
```

which will be the desired behavior in most cases.

8.2.1.4 Defining Argument Lists Any argument except the first, dispatched argument can be passed by association with a keyword rather than by its position. This feature is useful for long argument lists, particularly if many of the arguments are optional. The keyword arguments are separated from the positional arguments by the reserved word keywords in the definition of the argument list.

The arguments defined after the keywords reserved word are associated with the formal parameters by argument name, rather than by position. For example, the Part_display operation could be defined with additional parameters specifying color of the display and the rendering level.

```
Part_display (p:Part, keywords c: color, rlev: Integer)
```

Now invocations of the Part_display operation do not have to pass the arguments in order, so long as they are labeled with the argument names.

Keywords arguments can be declared as optional by including the reserved word optional in the argument list. If the definition above is changed to

```
Part_display (p:part,
keywords c:color, optional rlev: Integer);
```

then the rlev parameter does not have to be specified when the operation is invoked.

The action taken if rlev is not provided is determined by the operation itself, either by the use of the optional clause, or by testing the argument for validity inside the method implementing the operation. Normally, any attempt to access an optional argument which was not passed and does not have a default value will result in the NoValue exception. There are two exceptions to this rule, however: the argument can be explicitly tested for validity using the hasvalue construct or it can be used as an optional argument to another operation. If an optional argument is passed to another operation (where it must also be an optional argument) and the argument has no value in the caller, it also has no value in the called operation. Neither of these two accesses to the argument triggers the NoValue exception.

Default values may be provided for optional arguments in the same way that default values are assigned to properties in the properties statement. An argument with a default value is specified as

optional *name:type-specifier := expression*

For these arguments, the default value, as defined by the expression, is provided to the operation if no explicit value is furnished.

For instance, a default rendering level could be provided as

```
Part_display (p: Part,
keywords c:color, optional rlev: Integer :=1);
```

where the value 1 means that wire-frame rendering is the default.

8.2.2 Defining Exceptions

The exception handling mechanism consists of two parts, the raising of exceptions and the handling of exceptions. Raising is the way an operation notifies its caller of an exceptional condition; handling is the way the caller responds

to such notification.

An exception is an object that is created when one of a specified class of exceptional or unusual events occur, usually an error. The exception object is passed upwards through the flow of control until the relevant piece of code for handling the exception is found. The act of creating an exception object is called raising the exception while the activity of the exception handler code is known as catching the exception.

When an exception is raised, information about the context of the exceptional event is saved as the values of the properties of the exception, defined in the definition of the exception. There is a hierarchy of exception types (as with all other types), each member of which may augment its supertype's definition by defining additional properties. All exceptions, however, are subtypes of the exception type Failure.

Exceptions are defined in much the same fashion as types are: users can create their own exception definitions to cover particular exceptional events. A number of predefined exceptions are included in the system, for example

```
object Exception OutOfMemory is
  supertypes := Failure ;
  properties are
    AmtRequested : Integer;
  end properties ;
end OutOfMemory;
```

The only distinctive piece of information defined by this exception is the amount of memory the failed request was seeking. The handler of this exception could use this information to reduce the size of the request or to attempt a different approach to the problem. All exceptions are directly or indirectly a subtype of the exception Failure.

The exception specification of an operation can be changed by a refinement to exclude any exception condition mentioned in the original definition, or to add exceptions which are subtypes of exceptions mentioned in the original definition. For example, type Number defines the operation square_root, which has an Imaginary exception, which is raised when the input is negative. The type Complex refines square_root to exclude this exception.

User-defined exceptions are defined in the same way. For instance, the Part_display operation could have the following exceptions:

```
object type Part is
  . . .
  operations are
    Part_display (p: Part )
          method (shuttle_part_display)
          raises (NoImage) ;
  end operations;
end Part;
```

where the only exception raised is the NoImage exception. If an operation raises an exception, that exception must be defined to the system. The Failure exception is provided by Vbase, but the NoImage exception shown above must be defined by the user. For instance, there could be a general type of Part exception, of which NoImage is a subtype, as shown below:

```
object Exception PartError is
  supertypes := Failure ;
  properties are
    error_text : String ;
  end properties;
end PartError;
object Exception NoImage is
  supertypes := PartError ;
end NoImage;
```

The exception NoImage does not add any new properties, but exists only to distinguish the different cases of

PartError exceptions.

The property specification, as described above, translates directly into operation specifications on the operations which provide access to properties. The advantage of a property specification is that it is more compact. For example, the valuespec in a property definition translates into portions of the operation specifications of a number of different operations; this spec need only be stated once, and the system will propagate it to the correct operation specs.

8.2.3 Defining Iterators

An iterator is a special case of operations that provides a generalized looping behavior for aggregates of objects. It is specified much like an operation, except that no value is returned, but rather, there is a yield statement with similar syntax. A return statement returns a value of the specified type at termination of the operation. A yield statement, on the other hand, returns the current value of the specified type and suspends the operation, rather than terminating it.

Iterators may only be invoked as part of the language's **iterate** looping construct. An iterator thus invoked runs as a limited form of co-routine, processing as much as needed between loop iterations to supply a value for the next iteration of the loop. The loop terminates when the iterator returns (instead of yielding) or when the body of the loop exits. In all cases, the iterator invocation is terminated before control leaves the body of the loop.

An example of an iterator is in this definition of the type Aggregate:

```
object type Aggregate is
  ...
  operations are
    ...
    iterator Iterate (a:Aggregate)
                yields (e:Entity);
  end operations;
end Aggregate;
```

The operation Iterate is an iterator, as shown by the keyword **iterator**, and yields one value of type Entity, which, logically, should be an element of the aggregate.

The Iterate iterator for type Aggregate is provided by Vbase and, since all aggregates are a subtype of Aggregate, it is inherited by all aggregates. Users can define iterators as well, or refine the Iterate iterator if they create Aggregate subtypes.

As with all other operations, iterators are implemented with methods. There is one fundamental distinction between iterator methods and other methods: logically, the iterator method runs as a co-routine to the body of the loop.

When an iterate statement is executed, the iterator method is run until a yield statement is reached. The yield statement returns an element to the iterate statement, which assigns it to the iteration variable, and the body of the loop is executed. After the body is executed, control returns to the iterate statement. At this point, execution continues from the point immediately following the yield statement, with the local state preserved. Global state may change, but the flow of control and local state continue as if the yield statement were not present.

When the iterator method returns, the loop is terminated. Note that all iterator functions are null-valued (no return value is legal for return statements in an iterator method).

Iterators may also be exited by a **break** statement occurring within the loop body or by some other standard transfer of control out of the loop body. When such an exit occurs, it looks to the iterator as if the yield statement raised an exception. The exception provides the mechanism to unlock any locks that may have been gathered by the iterator and other such symmetric functions, through the use of protect statements.

Other exceptions raised by the execution of the loop body, and not caught within the loop body, also cause an exception to be raised in the co-routine. In general, the protect statement offers the safest way to guarantee that a piece of code is executed following a statement or upon termination of the loop. If an exception is raised in this way, the co-routine is aborted.

8.2.4 Defining Triggers

A method is bound to a trigger in a similar manner to any operation definition or refinement. The keyword **trigger** is used to identify the trigger method.

```
object type Part is
  ...
  properties are
    ECN: Eng_Doc inverse Eng_Doc$OfPart;
        define set
          method (Entity$genericSetmethod)
          triggers (check_ECN);
  end properties;
    ...
end Part;
```

Triggers allow the programmer to insert a method in the operation execution chain before or after any refinement or base definition of an operation. This is generally useful when the implementor of the trigger does not have or want access to the implementation details of the operation but does want some additional behavior to take place.

A trigger is attached to the definition of an operation or to one of its refinements. Its method is executed prior to the execution of the method for its host type. An operation can have multiple triggers.

8.2.5 Interfaces

Name contexts restrict name visibility, so access must be provided from one name context to names in another name context. This is accomplished through interfaces. Interfaces are simply aggregates of names. In a sense, one can think of name contexts as aggregates of names, too. Name contexts are distinguished from interfaces in that name contexts provide scoping. Interfaces merely provide a convenient way of grouping a set of names; they have no effect on scoping. Interfaces are themselves named objects, and can be used to establish the visibility of the names they contain in name contexts other than that which defines the interface. This is done by using the **import** statement as shown below,

```
object package Parts_Module is

  import Documents;
  ...

end Parts_Module;
```

where Documents is an interface defined elsewhere. The effect of the import statement is to cause all names contained within the Documents interface to become visible within the Parts_Module name context. The names visible in a name context are all names introduced by definition inside the name context plus all names inherited from containing name contexts combined with the names made visible by importing interfaces into the name context.

9. Conclusions

This section summarizes our conclusions and recommendations. We will evaluate the extent to which our approach supports extensibility, and the feasibility of building a prototype of the system for the Phase II effort. Our goal for Phase I was to examine concepts in the Object Manager which support extensibility. We wanted to document product requirements, and develop a concise, semi-formal specification for the software to be developed during Phase II.

9.1 Phase I Results

The objective of this initial effort was to examine and define the project and product requirements. An additional goal was to evaluate the feasibility of building the product (prototype in Phase II, market-ready version in Phase III). We pursued four activities:

- Examined design application needs to establish basic objectives and criteria for the system.
- Examined Ada language issues relevant to persistent object storage.
- Described an Ada/Object Oriented Data Base interface.
- Assessed the feasibility of building the interface.

Table 9-1 summarizes our conclusions about the effectiveness of this approach.

TABLE 9-1. Analysis Summary

Criterion	Analysis
Expressibility	Use integrated language to express complex modeling through **objects**, **types**, **operations**, and **properties**. Enforce semantic restrictions through *strong typing*.
Modularity	Separate specification/implementation. Cluster properties and operations into *abstract data type* descriptions.
Flexibility	Use Vbase access to dynamically create and access objects, types, operations and relationships. Handle exceptional conditions.
Efficiency	Separate specification/representation. Use Ada pragmas and representation specifications, Vbase storage pragmas.
Persistence	Rely on Vbase automated storage management facilities. Able to use Vbase's "DBMS Amenities" such as concurrency control.

9.1.1 Defined Issues and Identified Needs

We examined the particular needs of CAD/CAM developers, and identified their need for extensible databases. That study demonstrated that object-oriented technology offered an excellent solution to their problem. It also investigated the feasibility of combining the concepts of *modularity* and *abstraction* from programming languages; *persistent storage* from database systems; and *objects* from research prototypes.

We feel that Vbase, as an object-oriented database, embodies the necessary technology for database extensibility. The concepts of inheritance and refinement provide superior support for extensibility, with a database system providing performance potential to meet the most demanding needs of computer aided design support systems. The question, from our point of view, was whether Ada provided adequate and reasonable means to support access to persistent objects — in a well integrated and efficient manner.

9.1.2 Examined Ada

We examined Ada language issues from the perspective of adding persistent storage, using Vbase. Our focus during Phase I was not to solve all of the details of such an interface, but to explore the risk areas and assess the difficulties of the approach. Table 2 compares some of the pertinent features of Ada and Vbase. We decided building the interface is feasible, and that there are certain features we can exploit to integrate Ada with our product. However, there are a few areas which may pose unusual implementation difficulties:

TABLE 9-2. Summary of Features

Criterion	Ada Features	Vbase Features
Expressibility	Variables, types, subtypes packages, subprograms and exceptions	Object versions of same (plus iterators) as database entities
Modularity	Separation of specification and body	(Same)
Flexibility	*context clauses* (with)	import clauses. Run-time *method dispatching*
Efficiency	main memory representation clauses Compile-time optimization	Secondary storage pragmas Strong type checking
Persistence	package DIRECT_IO	Automatic storage management

- Integration of the type systems of Ada and Vbase, and the degree of type checking available,
- The difficulty of translating Vbase Data Model information into Ada package structures in the Ada program library,
- The compatibility of specific Vbase constructs (for example, **iterators** and **exceptions**) with Ada,
- The interaction of representation specifications, storage **pragmas**, and the Ada compiler.

9.1.3 Described Interface

We presented an approach for integrating Ada and Vbase and a description of the Ada/Vbase interface. The key element of this interface was a unified language for data definition, object manipulation, and application development. We selected Ada as an interface to Vbase for two reasons:

1. It integrates, in one language, much of the support for modularity and abstraction we support in Vbase.
2. Its standardization guarantees wide availability and a great deal of consistency between implementations — it is a reasonably stable interface.

We developed examples of Vbase entities which demonstrated the sort of database extensibility required by design applications. We built a parser for the interface language, and used it to check the consistency of our examples.

9.1.4 Analyzed Feasibility

Finally, Phase I analyzed the feasibility of actually *building* the interface. The implementation of the Vbase is based on the use of a preprocessor program to link the object-oriented system to a traditional programming language. The effect is to embed the functions of the database at both compile time and run time. We propose a similar form of implementation for the Ada interface.

The purpose of the Phase II effort is to explore the risk areas by building a prototype version of UNIVERS. We decided that the Phase II effort is feasible, and will result in a product which makes object-oriented technology available to a large market: the Government and its supporting contractors. The combination of Ada and Vbase provides a powerful system which meets the needs of design applications. The architecture we've defined presents the system in a flexible, yet consistent, package which should be easy to use for design application builders used to working with Ada.

9.2 Phase II Approach

Our goal for the UNIVERS product is to combine the expressibility of modern programming languages and the persistent storage management of databases with the inheritance/refinement mechanism of object-oriented systems to provide an extensible database system. The Phase II efforts implement the Ada/OODB interface defined during Phase I. Our objective for Phase II is to demonstrate a working Ada access to Vbase. We are concerned with three primary goals:

- To provide integrated support for persistent objects. The support should be as transparent as possible to users of the system.

- To make the system as type-safe as possible.
- To provide Ada access to the complete Vbase development system — including the database, an interactive object editor for traversal of type and object definitions, a verifier to check physical database consistency, and a debugger for operations/method examination

To accomplish these goals, we will pursue three major tasks:

- Designing and building the pre-processor
- Designing and building the Ada interface package
- Designing and building a small application which uses these two pieces to demonstrate the access to Vbase.

Table 9-3 summarizes Phase II of this project. We will build initial versions of the Pre-processor (described in Section 6) and the Interface Package (described in Section 7). We will also provide an application which demonstrates the working interface to Vbase. A Final Report will accompany the working software.

TABLE 9-3. Phase II Work Efforts

Deliverable	Work Efforts
Pre-Processor	Build on the Phase I recognizer to use the Vbase interface. Build the Ada package extractor.
Database Interface	Build Ada package(s) providing access to Vbase from the Ada program library.
Test Application	Build an application which uses the pre-processor, data model, and interface package to demonstrate extensibility.
Design	Document the design approach, including plans, "as built" specification, test & integration results

The work on the pre-processor will concentrate on the issue of type compatibility between Ada and Vbase. It will explore the difficulty of integrating their type systems to support our type checking goals for the product.

As part of the pre-processor, but as a separate work effort, we will prototype an Ada "package extractor". This effort will explore the difficulties of reflecting changes in Vbase state into the Ada program library. It will also examine the problems associated with automatic Ada code generation.

The prototype of the interface package will examine any issues of run-time library integration between Ada and Vbase. It will provide procedure-level access to the storage management layer of Vbase.

We will also build a small application to test the components of the prototype. It will demonstrate the degree to which we've achieved our goals.

A. Glossary

This sections defines terms we've introduced or referenced from the literature.

Abstraction The essence of abstraction is to extract essential properties while omitting inessential details.[Ros75]

Abstract Data Type A specification of the implementation of the abstract entities and operations which are encapsulated so that the only way to access or modify the entities is by the abstract operations.[Row80]

Allocator The evaluation of an allocator creates an *object* and returns a new *access value* which *designates* the object.[DOD83]

Compilation unit A compilation unit is the *declaration* or the *body* of a *program unit*, presented for compilation as an independent text. It is optionally preceded by a *context clause*, naming other compilation units upon which it depends.[DOD83]

Derived type A derived type is a *type* whose operations and values are replicas of those of an existing type. The existing type is called the *parent type* of the derived type.[DOD83]

Exception An exception is an error situation which may arise during program execution. To *raise* an exception is to abandon normal program execution so as to signal that the error has taken place. An *exception handler* is a portion of program text specifying a response to an exception. Execution of such a program text is called *handling* the exception.[DOD83]

Generic unit A generic unit is a template either for a set of *subprograms* or for a set of *packages*. A subprogram or package created using the template is called an *instance* of the generic unit. A *generic instantiation* is the kind of *declaration* that creates an instance.

A generic unit is written as a subprogram or package but with the specification prefixed by a *generic formal part* which may declare *generic formal parameters*. A generic formal parameter is either a *type*, a *subprogram*, or an *object*. A generic unit is one of the kinds of *program unit*.[DOD83]

Information Hiding A design decomposition method where every module is characterized by its knowledge of a design decision which it hides from all others.[Par72, Par71]

Object An object contains a value. A program creates an object either by *elaborating* and *object declaration* or by *evaluating* an *allocator*. The declaration or allocator specifies a *type* for the object: the object can only contain values of that type.[DOD83]

Overloading An identifier can have several alternative meanings at a given point in the program text: this property is called *overloading*. For example, an overloaded enumeration literal can be an identifier that appears in the definitions of two or more *enumeration types*. The effective meaning of an overloaded identifier is determined by the context. *Subprograms*, *aggregates*, *allocators*, and string *literals* can also be overloaded.[DOD83]

Package A package specifies a group of logically related entities, such as *types*, *objects* of those types, and *subprograms* with *parameters* of those types. It is written as a *package declaration* and a *package body*. The package declaration has a *visible part*, containing the *declarations* of all entities that can be explicitly used outside the package. It may also have a *private part* containing structural details that complete the specification of the visible entities, but which are irrelevant to the user of the package. The *package body* contains implementations of *subprograms* that have been specified in the package declaration. A package is one of the kinds of *program unit*.[DOD83]

Polymorphism In the context of object-oriented programming, polymorphism refers to the capability of different classes to respond to exactly the same message protocols, allowing interchangeable pieces as enabled by method sending.[Ste86]

Record type A value of a record type consists of *components* which are usually of different *types* or *subtypes*. For each component of a record value or record *object*, the definition of the record type specifies an identifier that uniquely determines the component within the record.[DOD83]

Representation clause A representation clause directs the compiler in the selection of the mapping of a *type* or an *object* onto features of the underlying machine that executes a program. In some cases, representation clauses completely specify the mapping; in other cases, they provide criteria for choosing a mapping.[DOD83]

Subprogram A subprogram is either a *procedure* or a *function*. A procedure specifies a sequence of actions and is invoked by a *procedure call* statement. A function specifies a sequence of actions and also returns a value called a *result*, and so a *function call* is an *expression*. A subprogram is written as a *subprogram declaration*, which specifies its *name*, *formal parameters*, and (for a function) its result; and a *subprogram body* which specifies the sequence of actions. The subprogram call specifies the *actual parameters* that are to be associated with the formal parameters. A subprogram is one of the kinds of *program unit*.[DOD83]

Subtype A subtype of a *type* characterizes a subset of the values of the type. The subset is determined by a *constraint* on the type. Each value in the set of values of a subtype *belongs* to the subtype and *satisfies* the constraint determining the subtype.[DOD83]

Trigger An object, a triggered action, and operations on the object that will, as a side effect, activate the triggered actions.[Ber87]

Type A type characterizes both a set of values, and a set of *operations* applicable to those values. A *type definition* is a language construct that defines a type.[DOD83]

Transaction An execution of one or more programs that should appear to be atomic relative to other transactions, in the sense that transactions appear to execute serially, and that each transaction either executes in its entirety or not at all.[Ber87]

Visibility At a given point in a program text, the *declaration* of an entity with a certain identifier is said to be *visible* if the entity is an acceptable meaning for an occurrence at that point of the identifier. The declaration is *visible* by *selection* at the place of a *selector* in a *selected component* or at the place of the name in a *named association*. Otherwise, the declaration is *directly visible*, that is if the identifier alone has that meaning.[DOD83]

B. Acronyms

AIO	An Instance Of
AKO	A Kind Of
APO	A Part Of
APSE	Ada Programming Support Environment
BSD	Berkeley Systems Distribution — one of the two main variants of the UNIX operating system.
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CASE	Computer Aided Software Engineering
CIM	Computer Integrated Manufacturing
DBMS	Data Base Management System
DDL	Data Definition Language
DML	Data Manipulation Language
ECAD	Electrical Computer Aided Design — CAD for electrical engineering design.
MCAD	Mechanical Computer Aided Design — CAD for mechanical engineering design.
MIPS	Million Instructions per Second — a measure of computer processing power
OODB	Object Oriented Data Base
VLSI	Very Large Scale Integration

C. References

Aho74 Aho, Alfred V. and Johnson, Steven C., "LR Parsing," *Computer Surveys*, vol. 6, no. 2, pp. 99-124, June 1974.

Aho75a Aho, Alfred V., Johnson, Steven C., and Ullman, Jeffrey D., "Deterministic Parsing of Ambiguous Grammars," *Communications of the ACM*, vol. 18, no. 8, pp. 441-452, Association for Computing Machinery, August 1975a.

Aho75b Aho, A. V. and Corasick, M. J., "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, pp. 333-340, Association for Computing Machinery, 1975b.

Aho77 Aho, Alfred. V. and Ullman, Jeffrey D., *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

Ber87 Bernstein, Philip A., "Database System Support for Software Engineering: An Extended Abstract," Technical Report TR-87-01, Wang Institute of Graduate Studies, Tyngsboro, MA, February, 1987.

Boo83 Booch, E. Grady, *Software Engineering with Ada*, Benjamin/Cummings, Reading, MA, 1983.

DOD83 DOD, U.S. Department of Defense, "Reference Manual for the Ada Programming Language," (ANSI/MIL-STD-1815A), American National Standards Institute, Inc., Washington, DC, February 17, 1983.

DOD86 DOD,, *FY-87 Defense Small Business Innovation Research Program (SBIR) Program Solicitation*, pp. 101-102, U.S. Department of Defense, Washington, DC, 1 October 1986.

Dah66 Dahl, O.-J. and Nygaard, K., "Simula — An Algol-based Simulation Language," *Communications of the ACM*, vol. 9, no. 9, pp. 671-678, Association for Computing Machinery, September 1966.

Gol83 Goldberg, Adele and Robson, David, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

Jac75 Jackson, Michael, *Principles of Program Design*, Academic Press, New York, NY, 1975.

Jac83 Jackson, Michael A., *System Development*, Prentice-Hall, 1983.

Joh75 Johnson, Steven C., "Yacc — Yet Another Compiler-Compiler," Computer Science Technical Report, Bell Laboratories, Murray Hill, NJ, July 1975.

Ker78 Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

Les75 Lesk, M. E., "Lex — A Lexical Analyzer Generator," Computer Science Technical Report, Bell Laboratories, Murray Hill, New Jersey, October 1975.

Lis74 Liskov, Barbara H. and Zilles, Steven N., "Programming with Abstract Data Types," *SIGPLAN Notices*, vol. 9, no. 4, Association for Computing Machinery, 1974.

Lis77 Liskov, Barbara, Snyder, Alan, Atkinson, Russell, and Schaffert, Craig, "Abstraction Mechanisms in CLU," *Communications of the ACM*, vol. 20, no. 8, pp. 564-576, Association for Computing Machinery, August 1977.

Lis86 Liskov, Barbara and Guttag, John, *Abstraction and Specification in Program Development*, MIT Electrical Engineering & Computer Science Series, MIT Press, Cambridge, MA, 1986.

McL80 McLeod, Dennis and Smith, John Miles, "Abstraction in Databases," in *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling*, pp. 19-22, ACM, June, 1980.

Par71 Parnas, David L., "Information Distribution Aspects of Design Methodology," CMU-CS- Tech. Report, Carnegie-Mellon University, Pittsburgh, PA, 1971.

Par72a Parnas, David L., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol. 15, no. 5, pp. 330-336, Association for Computing Machinery, May 1972.

Par72b Parnas, David L., "On the Criteria to be Used in Decomposing a System into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, Association for Computing Machinery, December 1972.

Par76 Parnas, David L., Handzel, Georg, and Wurges, Harald, "Design and Specification of the Minimal Subset of an Operating System Family," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 301-307, Institute of Electrical and Electronics Engineers, December 1976.

Ros75 Ross, Douglas T., Goodenough, John B., and Irvine, Charles A., "Software Engineering: Process, Principles, and Goals," *IEEE Computer*, pp. 65-67, Institute of Electrical and Electronics Engineers, May 1975.

Row80 Rowe, Lawrence A., "Data Abstraction From a Programming Language Viewpoint," in *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling*, pp. 29-35, ACM, June, 1980.

Str86 Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

Sow80 Sowa, John F., "A Conceptual Schema for Knowledge-Based Systems," in *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling*, pp. 193-195, ACM, June, 1980.

Ste86 Stefik, Mark and Bobrow, Daniel G., "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, vol. 6, no. 4, American Association for Artificial Intelligence, Winter 1986.

Vba86a Vbase,, "Vbase Data Model Reference Manual," Release 1.0, Ontologic, Inc., Billerica, MA, November 17, 1986.

Vba86b Vbase,, "Vbase TDL Language Reference Manual," Release 1.0, Ontologic, Inc., Billerica, MA, November 14, 1986a.

Vba86c Vbase,, "Vbase Functional Specification," Release 1.0, Ontologic, Inc., Billerica, MA, November 14, 1986b.

Vba87a Vbase,, "Vbase COP Language Reference Manual," Release 1.0, Ontologic, Inc., Billerica, MA, January 5, 1987a.

Vba87b Vbase,, "Vbase Technical Overview," Release 0.8, Ontologic, Inc., Billerica, MA, March 6, 1987b.

Wei81 Weinreb, D. and Moon, D., *The LISP Machine Reference Manual*, MIT AI Lab, 1981.

Wir76 Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, NJ, 1976.

Wir77 Wirth, Niklaus, "MODULA: A Language for Modular Multiprogramming," *Software - Practice and Experience*, vol. 7, 1977.

D. Grammar

The syntax is presented in a modified BNF using the following notational conventions:

bold	Terminal symbols
italic	Non-terminal symbols
[*X*]	Zero or one occurrence of *X*
{ *X* }	Zero or more occurrences of *X*

1. Declarations

Declarations describe objects and relationships. They also describe operations and object representation.

compilation ::= [*import-form*] *form*

form ::= *module-definition*
::= *entity-definition*
::= *type-definition*
::= *variable-definition*
::= *constant-definition*
::= *operation-definition*
::= *exception-definition*

module-definition ::= **object package** *defn-header* **is**
{ *module-form* }
[**private** { *type-form* }]
end [*db-name*] ;

entity-definition ::= **object** *db-simple-type db-name* **is**
{ *module-form* }
end [*db-name*] ;

type-definition ::= **object type** *defn-header* **is**
{ *type-form* }
end [*db-name*] ;

variable-definition ::= **object** *db-name* : *vspec-expression*
[:= *expression*] ;

constant-definition ::= **object** *db-name* : **constant** [*db-simple-type*]
:= *expression* ;

operation-definition ::= **object procedure** *db-name* [(*parameter-list*)]
[**raises** (*exception-list*)] **is**
{ *method-spec* }
{ *operation-form* }
{ *declaration* }
begin
{ *statement* }
end [*db-name*] ;

::= **object function** *db-name* [(*parameter-list*)]
return *db-simple-type*
[**raises** (*exception-list*)] **is**
{ *method-spec* }
{ *operation-form* }
{ *declaration* }
begin
{ *statement* }
end [*db-name*] ;

::= **method** *db-name* [(*parameter-list*)]
[**raises** (*exception-list*)] **is**
{ *method-spec* }
{ *operation-form* }
{ *declaration* }
begin
{ *statement* }
end [*db-name*] ;

::= **iterator** *db-name* [(*parameter-list*)]
yields *db-simple-type*
[**raises** (*exception-list*)] **is**
{ *method-spec* }
{ *operation-form* }
{ *declaration* }
begin
{ *statement* }
end [*db-name*] ;

exception-definition ::= **object exception** *db-name*
[(*parameter-keys*)] **is**
{ *exception-form* }
end [*db-name*] ;

declaration ::= **enter** *module-name* ;
::= *type-specifier* ;

import-form ::= **import** *db-name* { , *db-name* } ;

module-form ::= *form*
::= *initializer*

defn-header ::= *db-name* [(*parameter-list*)]

type-form ::= *module-form*
::= *instance-properties*
::= *instance-operations*
::= *representation*

instance-properties ::= **properties are**
{ *property-spec* }
end properties;

instance-operations ::= **operations are**
{ *operation-spec* }
end operations;

representation ::= **for representation use**
{ *rep-defn* }
end representation;

Instance Properties

property-spec ::= [**refines**] *db-name* :
[*property-kind*]
vspec-expression

[:= *expression*]
{ *property-option* }
;

property-kind ::= **optional**
::= **distributed**

property-option ::= *property-kind*
::= **allows** (*prop-acc* { , *prop-acc* })
::= **disallows** (*prop-acc* { , *prop-acc* })
::= **inverse** *db-name*
::= **unrefinable**
::= **constrainable** [**to** *vspec-expression*]
::= **object** *prop-op*
[**raises** (*exception-list*)]
{ *method-spec* }

prop-acc ::= **get**
::= **set**
::= **init**

prop-op ::= **get**
::= **set**

Instance Operations

operation-spec ::= [**refines**] **procedure** *identifier* [(*parameter-list*)]
[**raises** (*exception-list*)]
{ *method-spec* }
;
::= [**refines**] **function** *identifier* [(*parameter-list*)]
return (*return-list*)
[**raises** (*exception-list*)]
{ *method-spec* }
;
::= [**refines**] **iterator** *identifier* [(*parameter-list*)]
yields (*return-list*)
[**raises** (*exception-list*)]

{ *method-spec* }
;

parameter-list ::= [*parameter-spec* { , *parameter-spec* }]
[**keywords** *param-key-spec* { , *param-key-spec* }]

return-spec ::= [*identifier* =>] *vspec-expression*

exception-list ::= [*exception-name* { , *exception-name* }]

method-spec ::= **method** (*method-name*)
::= **triggers** (*method-name* { , *method-name* })

operation-form ::= *initializer*
::= *statement*

exception-form ::= *initializer*

method-name ::= *db-name*

parameter-spec ::= [**constrains**] *identifier* : *vspec-expression* [*constraint-spec*]

param-key-spec ::= **optional** *parameter-spec* [:= *expression*]
::= *parameter-spec*

constraint-spec ::= **constrainable** [**to** *vspec-expression*]

Representation

rep-defn ::= *identifier* : *type-expression* [*storage-layout*] ;

storage-layout ::= *cluster-clause*
::= (*storage-layout*)
::= *storage-layout* **and** *storage-layout*
::= *storage-layout* **or** *storage-layout*

cluster-clause ::= *cluster-type cluster-loc*

cluster-type ::= **filed**
::= **clustered**
::= **stored**

cluster-loc ::= **separately**
::= **with** *db-name* [(*expression*)]

2. Names

We have kept the semantically distinct name spaces of Ada and Vbase separate by distinguishing them syntactically. The $ lexical element is not used in Ada, and indicates a database object name.[1] We make this syntactic distinction even where the underlying semantics are very similar. For example, it will be impossible to define an Object Manager type as a subtype of some user-defined Ada type. However, it will be possible to assign values between objects of "compatible" types (for example, and Ada Integer and an object Integer). The $$ symbol has two uses: to anchor a database path name at the root name space in the database; and to the "next" method in a sequence (see the discussion of Statements).

1. In general, we have decided to use syntactic forms of resolving difficult semantic issues wherever possible. This may be a less palatable approach for language "purists," but saves us considerable effort in implementing a prototype in Phase II. We can certainly reconsider this decision *after* the prototype is working.

db-name ::= *identifier*
::= **$** *db-name*
::= *db-name* **$** *db-name*

interface-name ::= **$$** *module-name*
::= *module-name*

module-name ::= *identifier*
::= *module-name* **$$** *identifier*

type-specifier ::= **object** *obj-type*

obj-type ::= *identifier* [(*object-arg-list*)]

db-simple-type ::= *db-type-name* [(*argument-list*)]

3. Expressions

We have not settled the issues regarding an integration of expression elements in the two name spaces. For now, we will provide little implicit conversions between the Ada and Vbase types. The pre-processor will rely on the normal Ada overloading and subtype semantics to resolve ambiguities.

Some of the unary operators which appear to be "built-in" to the language will be transofrmed by the pre-processor into calls to the Vbase runtime.

initializer ::= *identifier* := *expression* ;
::= *identifier* := *aggregate* ;

vspec-expression ::= *type-expression*
::= **uniontype** (*vspec-expression* { , *vspec-expression* })
::= (*vspec-expression*)

type-expression ::= *db-simple-type*
::= **enum** (*enum-values*)
::= **record** (*record-fields*)
::= **variant** (*record-fields*)

primary-expression ::= **$** *identifier*
::= *interface-ref* **$** *identifier*
::= *primary-expression* ([*object-arg-list*])
::= **$$** (*object-arg-list*)
::= *primary-expression* . *identifier*
::= **assert** (*primary-expression* , *obj-type*)
::= *constructor*
::= **enable** (*primary-expression* , *obj-type*)

constructor ::= *identifier* **$(** *expression-list*)
::= *identifier* **$(** *index-value-list*)
::= *identifier* **$(** *key-value-list*)

object-arg-list ::= *expression* [, *object-arg-list*]
::= *identifier* : *expression* [*object-arg-list*]

argument-list ::= [*argument* { , *argument* }]

argument ::= *expression*
::= *identifier* => *expression*

4. Statements

The "statement" **$$** indicates the "next" method in sequence, which may be one of:

- A "trigger" in a sequence of triggers.
- A "base method" of the referenced operation.
- A method or trigger on the "supertype" operation.
- A method or trigger on an operation for which the referenced operation is a *refinement*.

All of these statements are transformed by the pre-processor into ANSI Ada, and ultimately realized by the subprograms in the database interface package. Some, such as **iterator,** pose interesting problems. We will explore these further in Phase II.

statement ::= **except** (*identifier* : *identifier*)
statement-list
end except;

::= **protect** *statement*

::= **raise** *identifier* [(*object-arg-list*)] ;

::= **reraise** *identifier* [(*object-arg-list*)] ;

::= **iterate** (*identifier* := *primary-expression*) *statement*

::= **iterate** (*identifier* := *identifier* (*object-arg-list*)) *statement*

::= **yield** (*expression*) ;

::= *call*

call ::= *db-name* [(*object-arg* { , *object-arg* })]

E. Software

This appendix describes the software of the parser which recognizes the language we've defined.

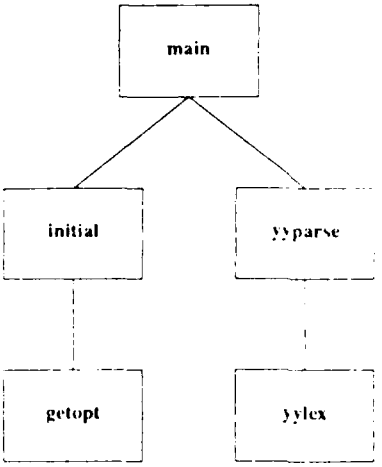


Figure E-1. Recognizer Function Names

Figure E-1 illustrates the software hierarchy, using the names of specific C functions.

1. Makefile

This file specifies the inter-file dependencies to the UNIX utility **make**. With this encoding of the software hierarchy, **make** can automatically generate new versions of the system incrementally -- by recompiling and relinking only those modules which have been modified.

```
NAME    = parser
OBJS    = lex.o tdl.o $(NAME).o   yyerror.o getopt.o
SRCS    = lex.l tdl.y $(NAME).c   yyerror.c  getopt.c
FILES   = makefile $(SRCS) $(NAME).1
YFLAGS = -vd
CFLAGS = -g -DYYDEBUG
LDFLAGS = -g

$(NAME) : $(OBJS)
          cc $(LDFLAGS) -o $(NAME) $(OBJS) -ll
tags      : makefile
          etags -t yyerror.c getopt.c yaccpar $(NAME).c
          touch tags

# Use the default 'make' rules for Yacc and Lex

lex.l : x.tab.h

tdl.o : y.tok.h yaccpar

x.tab.h: y.tab.h
          -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
```

2. Main Routine

The main routine uses **initial** and **getopt** to parse command line arguments and set the appropriate internal modes. It invokes the syntax analyzer through the call to **yyparse**.

```
/* $Header: parser.c,v 1.4 87/03/31 14:17:33 vilot Exp $ */
/* $Log:                parser.c,v $ Revision 1.4  87/03/31  14:17:33  vilot
 * Added `getopt' and debug option at the command line for `yydebug'
 *
 * Revision 1.3  87/03/02  11:08:29  vilot Minor changes.
 *
 * Revision 1.2  87/02/25  08:38:46  vilot Clean version {'old' TDL}
 *
 * Revision 1.1  87/02/20  17:29:13  vilot Initial revision
 *
 */
#include            <stdio.h>
#include            <strings.h>

extern int lines,                       /* Input line number (see: lex.l) */
extern int num_errors;                  /* number of errors found (see: lex.l) */
extern char *filename[];        /* buffer for filename in error message */

#ifdef YYDEBUG
extern int yydebug;

static int
initial (argc, argv)                                                    initial
int argc;
char *argv[]; {
    extern int getopt ();
    extern char *optarg;        /* getopt: option value access */
    extern int optind;                  /* getopt: will be index to 1st operand */
    int opterr = 0;                     /* number of erors */
    int flag;                           * option flag characters */
    char *optstring =                   /* getopt: string to be filled */
    "d";                                /* d = yydebug */
    char *usage =                       /* usage error message */
    "[-d] files";

    while ((flag = getopt (argc, argv, optstring)) != EOF) {
            switch (flag) {
            case 'd':
                yydebug = 1;
                break;
#ifdef 0
            case 't':
                traceflag = T;
                break;
#endif
            case '?':
            default:
                opterr++;
                break;
            }
    }
    if (opterr) {
            fprintf (stderr, "Usage: %s %s\n", argv[0], usage);
            exit (1);
    }
    return optind;
}

#endif YYDEBUG
```

main

```
main (argc, argv)                    /* parser: check Ada/TDL grammar */
int argc;
char *argv[];

{
    FILE *fp, *freopen ();
    int arg;                         /* index to 1st argument */
    int i;                           /* index over file arguments */

    arg = initial (argc, argv);

    if (argc == 1) {                 /* no args; copy standard input */
            strcpy (filename, "[stdin]");
            yyparse ();
    } else
            for (i = arg; i < argc; i++) {
                if ((fp = freopen (argv[i], "r", stdin)) == NULL) {
                        printf ("parser: can't open %s\n", argv[i]);
                        break;
                } else {
                        strcpy (filename, argv[i]);
                        lines = 1;
                        yyparse ();
                }
            }
    exit (num_errors);
}
```

3. Command Line Options

There is currently only one command line option defined — for enabling a debugging trace of the parsing actions.

getopt

```
/* $Header: getopt.c,v 1.2 87/01/27 09:53:20 vilot Exp $

 * Dependencies */

#include <stdio.h>

/* Globals */

/* get option letter from argument vector */
int opterr = 1,                         /* useless, never set or used */
    optind = 1,                         /* index into parent argv vector */
    optopt;                             /* character checked for validity */
char *optarg;                           /* argument associated with option */

#define BADCH       (int)'?'
#define EMSG        ""
#define tell(s)     fputs(*nargv,stderr);fputs(s,stderr); \
                    fputc(optopt,stderr);fputc('\n',stderr);return(BADCH);

int
getopt (nargc, nargv, ostr)
int nargc;
char **nargv, *ostr;
{
    static char *place = EMSG;              /* option letter processing */
    register char *oli;                     /* option letter list index */
    char *index ();

    if (!*place) {                          /* update scanning pointer */
        if (optind >= nargc || *(place = nargv[optind]) != '-' || !*++place)
            return (EOF);
        if (*place == '-') {  /* found "--" */
            ++optind;
            return (EOF);
        }
    }                                       /* option letter okay? */
    if ((optopt = (int) *place++) == (int) ':' || !(oli = index (ostr, optopt))) {
        if (!*place)
            ++optind;
        tell (": illegal option -- ");
    }
    if (*++oli != ':') {        /* don't need argument */
        optarg = NULL;
        if (!*place)
            ++optind;
    } else {                                /* need an argument */
        if (*place)
            optarg = place;  /* no white space */
        else if (nargc <= ++optind) {               /* no arg */
            place = EMSG;
            tell (": option requires an argument -- ");
        } else
            optarg = nargv[optind];                 /* white space */
        place = EMSG;
        ++optind;
    }
```

```
    return (optopt);                    /* dump back option letter */
}
```

4. Syntax Analyzer

This is the *yacc* input source, which produces a table-driven parser. The parser implements the language defined in Appendix D. It invokes the lexical analyzer via calls to **yylex**.

```
/* $Header: tdl.y,v 1.8 87/04/17 15:21:17 vilot Exp $ */
/*  $Log:                tdl.y,v $
 * Revision 1.8  87/04/17  15:21:17  vilot
 * This version passes all the simple tests, including nulltest.a,
 * constant.tdl, and package.tdl.
 *
 * Revision 1.7  87/04/17  15:11:48  vilot
 * This version passes all four "ds" tests: entity, property, operation,
 * and type .
 *
 * Revision 1.5  87/04/16  12:31:17  vilot
 * Added some more rules, refining  "form" ( 7 Apr 87 )
 *
 * Revision 1.4  87/03/31  14:16:33  vilot
 * This version agrees [at the top level] with Section 6 of the Report.
 *
 * Revision 1.3  87/03/02  11:07:15  vilot
 * Cleaned up grammar.  This version works ok, but has
 * shift/reduce errors due to optional constructs in 'Properties'
 *
 */
/*%terminals ---------------------------------------------------------------- */
%token identifier numeric_literal string_literal character_literal
%token ACCESS_ ALL_ AND_ ARRAY_ AT_
%token BEGIN_ BODY_
%token CASE_ CONSTANT_
%token DECLARE_ DO_
%token ELSE_ ELSIF_ END_ EXCEPTION_ EXIT_
%token FOR_ FUNCTION_
%token GENERIC_
%token IF_ IN_ IS_
%token LOOP_
%token MOD_
%token NEW_ NULL_
%token OF_ OR_ OTHERS_
%token PACKAGE_ PRIVATE_ PROCEDURE_
%token RAISE_ RECORD_ RETURN_
%token SELECT_ SUBTYPE_
%token THEN_ TYPE_
%token USE_
%token WHEN_ WHILE_ WITH_
%token ALLOWS_ ARE_
%token CLUSTERED_ CONSTRAINS_ CONSTRAINABLE_
%token DEFINE_ DISALLOWS_ DISTRIBUTED_
%token ENUM_
%token FILED_
%token IMPORT_ INVERSE_ ITERATOR_
%token KEYWORDS_
%token METHOD_
%token OBJECT_ OPERATIONS_ OPTIONAL_
%token PROPERTIES_
%token RAISES_ REFINES_ REPRESENTATION_ RETURNS_
%token SEPARATELY_ STORED_
%token TO_ TRIGGERS_ TYPE_PROPERTIES_
%token UNIONTYPE_ UNREFINABLE_
%token VARIANT_
%token YIELDS_
%token ENTER_ EXCEPT_ GETREP_ ITERATE_ MAKEREP_ PROTECT_ RERAISE_ YIELD_
%token ARROW_ DBLDOT_ EXP_ ASSIGN_ NOTEQL_ GTEQL_ LTEQL_ L_LBL_ R_LBL_ BOX_
```

```
%start compilation
%%
/* Ada  */
/* 10.1 */
compilation                         /* compilation_unit { compilation_unit } */
            : compilation_unit
            | compilation compilation_unit              { yyerrok; }
/*            | error */
/*            | compilation error */
            ;
compilation_unit
            : ..import_form.. form
            ;
/* TDL */
form
            : module_definition
            | entity_definition
            | type_definition
            | variable_definition
            | constant_definition
            | operation_definition
            | exception_definition
            ;
module_definition
            : OBJECT_ PACKAGE_ defn_header IS_
              ..module_form..
              ..private..
              defn_end
 ..private..
            : /* empty */
            | module_private                    { yyerrok; }
            ;
module_private
            :  PRIVATE_  /* ..type_form.. */
            ;
entity_definition
            : OBJECT_ db_simple_type defn_header IS_
              ..module_form..
              defn_end
            ;
type_definition
            : OBJECT_ TYPE_ defn_header IS_
              ..type_form..
              defn_end
            ;
variable_definition                 /* vspec-expression */
            : OBJECT_ db_name ':' identifier opt_expression ';'
            ;
constant_definition
            : OBJECT_ db_name ':' CONSTANT_ opt_db_type init_expression ';'
            ;
opt_db_type
            : /* null */
            | db_simple_type
            ;
operation_definition
            : procedure_definition
            | function_definition
            | method_definition
            | iterator_definition
            ;
procedure_definition
            : OBJECT_ PROCEDURE_ defn_header opt_raises
              IS_
              ..method_spec.. ..operation_form.. ..declaration..
```

```
                BEGIN_
                ..statement..
                defn_end
function_definition
            : OBJECT_ FUNCTION_ defn_header
              RETURN_ db_simple_type opt_raises IS_
              ..method_spec.. ..operation_form.. ..declaration..
              BEGIN_
              ..statement..
              defn_end
iterator_definition
            : ITERATOR_ defn_header
              YIELDS_ db_simple_type opt_raises IS_
              ..method_spec.. ..operation_form.. ..declaration..
              BEGIN_
              ..statement..
              defn_end
method_definition
            : METHOD_ defn_header opt_raises
              IS_ ..declaration..
             /* BEGIN_ */
             /* TBS */
              defn_end
            ;
exception_definition
            : OBJECT_ EXCEPTION_ defn_header IS_
              ..exception_form..
              defn_end
            ;
opt_raises
            : /* null */
            | RAISES_ param_LP ..exception_list.. param_RP
            ;
defn_end
            : END_ opt_db_name defn_semi
            ;
defn_semi
            : ';'                   { yyerrok; }
            | error ';'
            ;
..declaration..          /* { declaration } */
            : /* empty */
            | ..declaration.. declaration                   { yyerrok; }
            | ..declaration.. error
            ;
declaration
            : ENTER_ module_name decl_semi
            | type_specifier     decl_semi
            ;
decl_semi
            : ';'                   { yyerrok; }
            | error ';'
            ;
..import_form..          /* { import_form } */
            : /* empty */
            | ..import_form.. import_form                   { yyerrok; }
            | ..import_form.. error
            ;
import_form
            : IMPORT_ import_list ';'
            ;
import_list                     /* interface_name { ',' interface_name } */
            : interface_name
            | import_list ',' interface_name                        { yyerrok; }
            | error
```

```
            | import_list error
            | import_list error interface_name                    { yyerrok; }
            | import_list ',' error
            ;
 ..module_form..          /* module_form { module_form } */
            : module_form
            | ..module_form.. module_form                         { yyerrok; }
            | error
            | ..module_form.. error
            ;
module_form
            : form
            | initializer
            ;
defn_header
            : db_name opt_param_list
            ;
 ..type_form..            /* type_form { type_form } */
            : type_form
            | ..type_form.. type_form                             { yyerrok; }
            | error
            | ..type_form.. error
            ;
type_form
            : module_form
            | instance_properties
            | instance_operations
            | representation
            ;
instance_properties
            : PROPERTIES_ ARE_
              ..property_spec..
              END_ PROPERTIES_ ';'
            ;
instance_operations
            : OPERATIONS_ ARE_
              ..operation_spec..
              END_ OPERATIONS_ ';'
            ;
representation
            : FOR_ REPRESENTATION_ USE_
              ..rep_defn..
              END_ REPRESENTATION_ ';'
/* INSTANCE PROPERTIES */
 ..property_spec..  /* property_spec { property_spec } */
            : property_spec
            | ..property_spec.. property_spec                     { yyerrok; }
            | error
            | ..property_spec.. error
            ;
property_spec
            : opt_refines db_name ':' opt_property_kind
              value_spec opt_expression
              ..property_option.. ';'
            ;
opt_refines
            :  /* empty null */
            |  REFINES_
            ;
opt_property_kind
            : /* null */
            | property_kind
            ;
property_kind
            : OPTIONAL_
```

```
            | DISTRIBUTED_
            ;
 ..property_option../* { property_option } */
            : /* null */
            | ..property_option.. property_option                    { yyerrok; }
            | ..property_option.. error
            ;
property_option
            : property_kind
            | ALLOWS_       '(' ..prop_acc.. ')'
            | DISALLOWS_    '(' ..prop_acc.. ')'
            | INVERSE_ db_name
            | UNREFINABLE_
            | constraint_spec
            | OBJECT_ prop_op opt_raises ..method_spec..
            ;
 ..prop_acc..      /* prop-acc { ',' prop-acc } */
            : prop_acc
            | ..prop_acc.. ',' prop_acc                              { yyerrok; }
            | error
            | ..prop_acc.. error
            | ..prop_acc.. error prop_acc                            { yyerrok; }
            | ..prop_acc.. ',' error
            ;
prop_acc
            :  db_name              /* GET_ / SET_ / INIT_ */
            ;
prop_op
            : db_name               /* GET_ / SET_ */
            ;
/* INSTANCE OPERATIONS */
 ..operation_spec.. /* operation { operation } */
            : operation
            | ..operation_spec.. operation                           { yyerrok; }
            | error
            | ..operation_spec.. error
            ;
operation
            : opt_refines PROCEDURE_ db_name opt_param_list
              /* opt_raises */
               ..method_spec..
               ';'
            | opt_refines FUNCTION_  db_name opt_param_list
              RETURN_ db_name
              /* opt_raises */
               ..method_spec..
               ';'
            | opt_refines ITERATOR_  db_name opt_param_list
              YIELDS_ db_name
              /* opt_raises */
               ..method_spec..
               ';'
            ;
opt_param_list
            : /* null */
            | param_list
            ;
param_list
            : param_LP parameter_list param_RP
            ;
param_LP
            : '('                   { yyerrok; }
            ;
param_RP
            : ')'                   { yyerrok; }
```

```
            | error ')'
            ;
parameter_list
            : ..parameter_spec..
            | KEYWORDS_ ..param_key_spec..
            | ..parameter_spec.. KEYWORDS_ ..param_key_spec..
            ;
return_spec
            : expr_argument
            ;
 ..exception_list.. /* exception_name { ',' exception_name } */
/* error cases handled by param_RP */
            : ..exception_list.. ',' exception_name                    { yyerrok; }
            | error
/*            | ..exception_list.. error */
            | ..exception_list.. error exception_name                          { yyerrok; }
/*            | ..exception_list.. ',' error */
            ;
 ..method_spec..            /*  { method_spec } */
            : /* null */
            | ..method_spec.. method_spec              { yyerrok; }
/*            | ..method_spec.. error */
            ;
method_spec
            : METHOD_ '(' method_name ')'
            | TRIGGERS_ '(' ..method_name.. ')'
            ;
 ..method_name.. /* method_name { ',' method_name } */
            : method_name
            | ..method_name.. ',' method_name                          { yyerrok; }
            | error
            | ..method_name.. error
            | ..method_name.. error method_name                        { yyerrok; }
            | ..method_name.. ',' error
            ;
 ..operation_form../* operation_form { operation_form } */
            : operation_form
            | ..operation_form.. operation_form                        { yyerrok; }
            | error
/*            | ..operation_form.. error */
            ;
operation_form
            : initializer
            | statement
            ;
 ..exception_form../* exception_form { exception_form } */
            : exception_form
            | ..exception_form.. exception_form                        { yyerrok; }
            | error
            | ..exception_form.. error
            ;
exception_form
            : initializer
            ;
method_name
            : db_name
            ;
 ..parameter_spec../* parameter_spec { ';' parameter_spec } */
            : parameter_spec
            | ..parameter_spec.. ';' parameter_spec                    { yyerrok; }
/*            | error */
/*            | ..parameter_spec.. error */
```

```
/*          | ..parameter_spec.. error parameter_spec                    { yyerrok; } */
/*          | ..parameter_spec.. ';' error */
            ;
parameter_spec
            : opt_constrains db_name ':' value_spec opt_constraint_spec
            ;
opt_constrains
            : /* null */
            | CONSTRAINS_
            ;
 ..param_key_spec..        /* param_key_spec { ';' param_key_spec } */
            : param_key_spec
            | ..param_key_spec.. ';' param_key_spec                  { yyerrok; }
/*          | error */
/*          | ..param_key_spec.. error */
/*          | ..param_key_spec.. error param_key_spec                  { yyerrok; } */
/*          | ..param_key_spec.. ';' error */
            ;
param_key_spec
            : OPTIONAL_ parameter_spec opt_expression
            | parameter_spec
            ;
opt_constraint_spec
            : /* null */
            | constraint_spec
            ;
constraint_spec
            : CONSTRAINABLE_ opt_to_clause
            ;
opt_to_clause
            : /* null */
            | TO_ value_spec
            ;

/* REPRESENTATION */
 ..rep_defn..              /* { rep_defn } */
            : /* null */
            | ..rep_defn.. rep_defn          { yyerrok; }
            | ..rep_defn.. error
            ;
rep_defn
            : db_name ':' type_expression opt_storage_layout ';'
            ;
opt_storage_layout
            : /* null */
            | storage_layout
            ;
storage_layout
            : cluster_clause
            | '(' storage_layout ')'
/*          | storage_layout storage_choice storage_layout */
            ;
/* storage_choice : AND_ | OR_ ; */
cluster_clause
            : cluster_type cluster_loc
            ;
cluster_type
            : FILED_
            | CLUSTERED_
            | STORED_
            ;
cluster_loc
            : SEPARATELY_
```

```
            | WITH_ db_name /* '(' expression ')' */
            ;

/* NAMES */
opt_db_name
            : /* null */
            | db_name
/*              $db-name is in lex */
            ;
exception_name
            : identifier
            ;
db_name
            : identifier
/*            | TYPE_              /* for type "type" */
/*            | METHOD_            /* for type "method" */
/*            | PROCEDURE_         /* for type "procedure" */
/*            | EXCEPTION_         /* for type "exception" */
/*            | RETURNS_           /* a property of "operation" */
/*            | TRIGGERS_          /* a property of "operation" */
            ;
interface_name
            : identifier
            ;
module_name
            : identifier
            ;
type_specifier
            : OBJECT_ obj_type
            ;
obj_type
            : identifier opt_object_arg_list
            ;
db_simple_type
            : db_name
            ;

/* EXPRESSIONS */
/* initializer */
initializer
            :  db_name keyword_marker expression      initializer_semi
            |  db_name keyword_marker agg_expression initializer_semi
            ;
keyword_marker
            : IS_
            | error IS_
            | ARE_
            | error ARE_
            | ASSIGN_
            | error ASSIGN_
            ;
initializer_semi
            :   ';'
            |   error ';'
            ;
value_spec
            :  expression
            ;
type_expr_list_req          :  type_expression
                            |  type_expr_list_req ',' type_expression
                            ;
type_expression
            : expression
            | ENUM_     '(' tgen_enum_list_req ')'
            | RECORD_   '(' tgen_record_fields ')'
```

```
        | VARIANT_ '(' tgen_record_fields_req ')'
        ;
tgen_enum_list_req      : tgen_enum
                        | tgen_enum_list_req ',' tgen_enum
                        ;
tgen_enum               : db_name  tgen_enum_opt_expr
                        ;
tgen_enum_opt_expr      : /* empty null */
                        | '=' expression
                        ;
tgen_record_fields      :
                        | tgen_record_fields_req ';'
                        ;
tgen_record_fields_req  : tgen_record_field
                        | tgen_record_fields_req ';' tgen_record_field
                        ;
tgen_record_field       : db_name  tgen_record_embedded ':' value_spec
                        ;
tgen_record_embedded    : /* empty null */
                        | db_name
                        ;
/* primary_expression */

init_expression
        : ASSIGN_ expression
        ;
opt_expression
        : /* empty null */
        | init_expression
        ;
expression
        : literal
        | db_name
/*                        / expr_construct */
                        | expr_unaryMinus
                        | expr_precedence
/*                        / expr_range */
                        | type_parameterized
                        | vspec_union
/*                        / vspec_parameterized */
        ;
literal
        : numeric_literal
        | string_literal
        | character_literal
        ;
/*
expr_construct          : opt_expression DLP expr_argument_list ')'
                        | opt_expression DLB expr_argument_list ']'
                        ;
*/
expr_unaryMinus         : '-' expression
                        ;
expr_precedence         : '(' expression ')'
                        ;
/* expr_range              : expression DBLDOT_ expression */
type_parameterized      /* 1 shift/reduce */
        : identifier '(' expr_argument_list ')'
        ;
vspec_union
        : UNIONTYPE_ '(' type_expr_list_req ')'
        ;
```

```
/* vspec_parameterized          : db_name '[' expr_range ']' . */
/* constructor */
/* unary_expression */
/* argument_list */
/* argument */
opt_object_arg_list
        : /* null */
        | '(' object_arg_list ')'
        ;
object_arg_list                 /* expression { ',' obbject_arg_list } */
        : expression
        | expression ',' object_arg_list
        | identifier ':' expression
        | identifier ':' expression object_arg_list
        ;
opt_arg_list
        : /* null */
        | '(' obj_arg_list ')'
        ;
obj_arg_list                    /* { argument { ',' argument } */
        : expr_argument
        | obj_arg_list ',' expr_argument                    { yyerrok; }
        | error
        | obj_arg_list error
        | obj_arg_list error expr_argument                  { yyerrok; }
        | obj_arg_list ',' error
        ;
expr_argument
        :  expression
        |  named_association
        ;
named_association
        : db_name ARROW_ expression
        ;


/* STATEMENTS */
 ..statement..          /* statement { statement } */
        : statement
        | ..statement.. statement                   { yyerrok; }
        | error
        | ..statement.. error
        ;
statement
        : NULL_ ';'             /* for Testing only */
        | EXCEPT_ '(' identifier ':' identifier ')'
          ..statement..
          END_ EXCEPT_ ';'
        | PROTECT_ statement
        | RAISE_ identifier opt_arg_list ';'
        | RERAISE_ identifier opt_arg_list ';'
        | ITERATE_ '(' identifier ASSIGN_ /*primary_*/expression ')' statement
        | YIELD_ expression ';'
        | RETURN_ expression ';'
        | call
        ;
call
        : db_name opt_object_arg_list ';'
/*        / getrep */
/*        / makerep */
        ;
%%
```

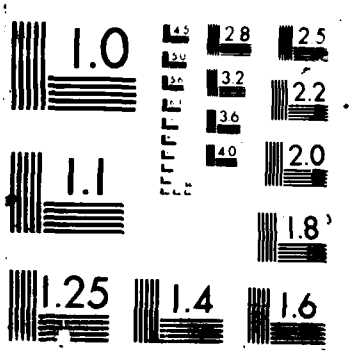
5. Lexical Analyzer

This is the *lex* input source, which produces a regular expression recognizer. Its basic function is to recognize keywords in the input stream.

```
%{
/* $Header: lex.l,v 1.4 87/04/17 17:06:42 vilot Exp $ */
/*  $Log:                    lex.l,v $
 * Revision 1.4  87/04/17  17:06:42  vilot
 * Added COP keywords, '$' to identifiers for db-names.
 *
 * Revision 1.3  87/03/02  11:11:18  vilot
 * Minor changes for TDL.
 *
 * Revision 1.2  87/02/25  08:36:00  vilot
 * Clean version {'old' TDL}.
 *
 * Revision 1.1  87/02/23  13:44:18  vilot
 * Initial revision
 *
 */
#include        "y.tab.h"

#define MAXLEN 256

int     lines = 1;                      /* line number being processed */
int         num_errors = 0;             /* number of errors encountered*/
char        *filename[MAXLEN];          /* buffer for filename in error message *

/*          #define linenum printf("\n[%d]\t",++lines)                  */
%}

%e      1500    /* tree nodes array size     */
%p      5500    /* positions                 */
%a      5500    /* transitions array size    */
%k      1500    /* packed character classes */
%o      7500    /* output array size         */

%START IDENT Z

a       [aA]
b       [bB]
c       [cC]
d       [dD]
e       [eE]
f       [fF]
g       [gG]
h       [hH]
i       [iI]
j       [jJ]
k       [kK]
l       [lL]
m       [mM]
n       [nN]
o       [oO]
p       [pP]
q       [qQ]
r       [rR]
s       [sS]
t       [tT]
u       [uU]
v       [vV]
w       [wW]
x       [xX]
```

AD-A187 405 UNIVERS PRODUCT PHASE 1(U) ONTOLOGIC INC BILLERICA MA 2/2
M J VILOT ET AL. 27 APR 87 N00014-86-C-0605

UNCLASSIFIED F/G 12/7 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

```
y       [yY]
z       [zZ]

%%
{a}{c}{c}{e}{s}{s}                  {ECHO; BEGIN Z; return(ACCESS_);}
{a}{l}{l}                           {ECHO; BEGIN Z; return(ALL_);}
{a}{n}{d}                           {ECHO; BEGIN Z; return(AND_);}
{a}{r}{e}                           {ECHO; BEGIN Z; return(ARE_);}
{a}{r}{r}{a}{y}                     {ECHO; BEGIN Z; return(ARRAY_);}
{a}{t}                              {ECHO; BEGIN Z; return(AT_);}
{b}{e}{g}{i}{n}                     {ECHO; BEGIN Z; return(BEGIN_);}
{b}{o}{d}{y}                        {ECHO; BEGIN Z; return(BODY_);}
{c}{a}{s}{e}                        {ECHO; BEGIN Z; return(CASE_);}
{c}{o}{n}{s}{t}{a}{n}{t} {ECHO; BEGIN Z; return(CONSTANT_);}
{d}{e}{c}{l}{a}{r}{e}               {ECHO; BEGIN Z; return(DECLARE_);}
{d}{o}                              {ECHO; BEGIN Z; return(DO_);}
{e}{l}{s}{e}                        {ECHO; BEGIN Z; return(ELSE_);}
{e}{l}{s}{i}{f}                     {ECHO; BEGIN Z; return(ELSIF_);}
{e}{n}{d}                           {ECHO; BEGIN Z; return(END_);}
{e}{x}{c}{e}{p}{t}{i}{o}{n} {ECHO; BEGIN Z; return(EXCEPTION_);}
{e}{x}{i}{t}                        {ECHO; BEGIN Z; return(EXIT_);}
{f}{o}{r}                           {ECHO; BEGIN Z; return(FOR_);}
{f}{u}{n}{c}{t}{i}{o}{n} {ECHO; BEGIN Z; return(FUNCTION_);}
{g}{e}{n}{e}{r}{i}{c}               {ECHO; BEGIN Z; return(GENERIC_);}
{i}{f}                              {ECHO; BEGIN Z; return(IF_);}
{i}{n}                              {ECHO; BEGIN Z; return(IN_);}
{i}{s}                              {ECHO; BEGIN Z; return(IS_);}
{l}{o}{o}{p}                        {ECHO; BEGIN Z; return(LOOP_);}
{m}{o}{d}                           {ECHO; BEGIN Z; return(MOD_);}
{n}{e}{w}                           {ECHO; BEGIN Z; return(NEW_);}
{n}{u}{l}{l}                        {ECHO; BEGIN Z; return(NULL_);}
{o}{b}{j}{e}{c}{t}                  {ECHO; BEGIN Z; return(OBJECT_);}
{o}{f}                              {ECHO; BEGIN Z; return(OF_);}
{o}{r}                              {ECHO; BEGIN Z; return(OR_);}
{o}{t}{h}{e}{r}{s}                  {ECHO; BEGIN Z; return(OTHERS_);}
{p}{a}{c}{k}{a}{g}{e}               {ECHO; BEGIN Z; return(PACKAGE_);}
{p}{r}{i}{v}{a}{t}{e}               {ECHO; BEGIN Z; return(PRIVATE_);}
{p}{r}{o}{c}{e}{d}{u}{r}{e} {ECHO; BEGIN Z; return(PROCEDURE_);}
{r}{a}{i}{s}{e}                     {ECHO; BEGIN Z; return(RAISE_);}
{r}{e}{c}{o}{r}{d}                  {ECHO; BEGIN Z; return(RECORD_);}
{r}{e}{t}{u}{r}{n}                  {ECHO; BEGIN Z; return(RETURN_);}
{s}{e}{l}{e}{c}{t}                  {ECHO; BEGIN Z; return(SELECT_);}
{s}{u}{b}{t}{y}{p}{e}               {ECHO; BEGIN Z; return(SUBTYPE_);}
{t}{h}{e}{n}                        {ECHO; BEGIN Z; return(THEN_);}
{t}{y}{p}{e}                        {ECHO; BEGIN Z; return(TYPE_);}
{u}{s}{e}                           {ECHO; BEGIN Z; return(USE_);}
{w}{h}{e}{n}                        {ECHO; BEGIN Z; return(WHEN_);}
{w}{h}{i}{l}{e}                     {ECHO; BEGIN Z; return(WHILE_);}
{w}{i}{t}{h}                        {ECHO; BEGIN Z; return(WITH_);}

{a}{l}{l}{o}{w}{s}                          {ECHO; BEGIN Z; return(ALLOWS_);}
{c}{l}{u}{s}{t}{e}{r}{e}{d}                 {ECHO; BEGIN Z; return(CLUSTERED_);}
{c}{o}{n}{s}{t}{r}{a}{i}{n}{s}              {ECHO; BEGIN Z; return(CONSTRAINS_);}
{c}{o}{n}{s}{t}{r}{a}{i}{n}{a}{b}{l}{e} {ECHO; BEGIN Z; return(CONSTRAINABLE_);}
{d}{e}{f}{i}{n}{e}                          {ECHO; BEGIN Z; return(DEFINE_);}
{d}{i}{s}{a}{l}{l}{o}{w}{s}     {ECHO; BEGIN Z; return(DISALLOWS_);}
{d}{i}{s}{t}{r}{i}{b}{u}{t}{e}{d} {ECHO; BEGIN Z; return(DISTRIBUTED_);}
{e}{n}{u}{m}                                {ECHO; BEGIN Z; return(ENUM_);}
{f}{i}{l}{e}{d}                             {ECHO; BEGIN Z; return(FILED_);}
{i}{m}{p}{o}{r}{t}                          {ECHO; BEGIN Z; return(IMPORT_);}
{i}{n}{v}{e}{r}{s}{e}                       {ECHO; BEGIN Z; return(INVERSE_);}
{i}{t}{e}{r}{a}{t}{o}{r}                    {ECHO; BEGIN Z; return(ITERATOR_);}
{k}{e}{y}{w}{o}{r}{d}{s}                    {ECHO; BEGIN Z; return(KEYWORDS_);}
{m}{e}{t}{h}{o}{d}                          {ECHO; BEGIN Z; return(METHOD_);}
{o}{p}{e}{r}{a}{t}{i}{o}{n}{s}              {ECHO; BEGIN Z; return(OPERATIONS_);}
```

```
{o}{p}{t}{i}{o}{n}{a}{l}                    {ECHO; BEGIN Z; return(OPTIONAL_);}
{p}{r}{o}{p}{e}{r}{t}{i}{e}{s}              {ECHO; BEGIN Z; return(PROPERTIES_);}
{r}{a}{i}{s}{e}{s}                          {ECHO; BEGIN Z; return(RAISES_);}
{r}{e}{f}{i}{n}{e}{s}                       {ECHO; BEGIN Z; return(REFINES_);}
{r}{e}{p}{r}{e}{s}{e}{n}{t}{a}{t}{i}{o}{n} {ECHO; BEGIN Z; return(REPRESENTATION_);}
{r}{e}{t}{u}{r}{n}{s}                       {ECHO; BEGIN Z; return(RETURNS_);}
{s}{e}{p}{a}{r}{a}{t}{e}{l}{y}              {ECHO; BEGIN Z; return(SEPARATELY_);}
{s}{t}{o}{r}{e}{d}                          {ECHO; BEGIN Z; return(STORED_);}
{t}{r}{i}{g}{g}{e}{r}{s}                    {ECHO; BEGIN Z; return(TRIGGERS_);}
{t}{y}{p}{e}{p}{r}{o}{p}{e}{r}{t}{i}{e}{s} {ECHO; BEGIN Z; return(TYPE_PROPERTIES_);}
{u}{n}{i}{o}{n}{t}{y}{p}{e}                 {ECHO; BEGIN Z; return(UNIONTYPE_);}
{u}{n}{r}{e}{f}{i}{n}{a}{b}{l}{e}  {ECHO; BEGIN Z; return(UNREFINABLE_);}
{v}{a}{r}{i}{a}{n}{t}                       {ECHO; BEGIN Z; return(VARIANT_);}
{y}{i}{e}{l}{d}{s}                          {ECHO; BEGIN Z; return(YIELDS_);}

{e}{n}{t}{e}{r}                     {ECHO; BEGIN Z; return(ENTER_);}
{e}{x}{c}{e}{p}{t}                  {ECHO; BEGIN Z; return(EXCEPT_);}
{i}{t}{e}{r}{a}{t}{e}               {ECHO; BEGIN Z; return(ITERATE_);}
{p}{r}{o}{t}{e}{c}{t}               {ECHO; BEGIN Z; return(PROTECT_);}
{r}{e}{r}{a}{i}{s}{e}               {ECHO; BEGIN Z; return(RERAISE_);}
{t}{o}                              {ECHO; BEGIN Z; return(TO_);}
{y}{i}{e}{l}{d}                     {ECHO; BEGIN Z; return(YIELD_);}

"=>"                                {ECHO; BEGIN Z; return(ARROW_);}
".."                                {ECHO; BEGIN Z; return(DBLDOT_);}
"**"                                {ECHO; BEGIN Z; return(EXP_);}
":="                                {ECHO; BEGIN Z; return(ASSIGN_);}
"/="                                {ECHO; BEGIN Z; return(NOTEQL_);}
">="                                {ECHO; BEGIN Z; return(GTEQL_);}
"<="                                {ECHO; BEGIN Z; return(LTEQL_);}
"<<"                                {ECHO; BEGIN Z; return(L_LBL_);}
">>"                                {ECHO; BEGIN Z; return(R_LBL_);}
"<>"                                {ECHO; BEGIN Z; return(BOX_);}

"&"                                 {ECHO; BEGIN Z; return('&'); }
"("                                 {ECHO; BEGIN Z; return('('); }
")"                                 {ECHO; BEGIN IDENT; return(')'); }
"*"                                 {ECHO; BEGIN Z; return('*'); }
"+"                                 {ECHO; BEGIN Z; return('+'); }
","                                 {ECHO; BEGIN Z; return(','); }
"-"                                 {ECHO; BEGIN Z; return('-'); }
"."                                 {ECHO; BEGIN Z; return('.'); }
"/"                                 {ECHO; BEGIN Z; return('/'); }
":"                                 {ECHO; BEGIN Z; return(':'); }
";"                                 {ECHO; BEGIN Z; return(';'); }
"<"                                 {ECHO; BEGIN Z; return('<'); }
"="                                 {ECHO; BEGIN Z; return('='); }
">"                                 {ECHO; BEGIN Z; return('>'); }
"|"|"!"                             {ECHO; BEGIN Z; return('|'); }
<IDENT>\'                           {ECHO; BEGIN Z; return('\'');}   /* type mark only */

"$"                                 {ECHO; BEGIN Z; return('$'); }
"["                                 {ECHO; BEGIN Z; return('['); }
"]"                                 {ECHO; BEGIN Z; return(']'); }
"{"                                 {ECHO; BEGIN Z; return('{'); }
"}"                                 {ECHO; BEGIN Z; return('}'); }

[\$a-z_A-Z][a-z_A-Z0-9\$]*          {ECHO; BEGIN IDENT; return(identifier);}
[0-9][0-9_]*([.][0-9_]+)?([Ee][-+]?[0-9_]+)?  {
                                    ECHO; BEGIN Z; return(numeric_literal);}

[0-9][0-9_]*#[0-9a-fA-F_]+([.][0-9a-fA-F_]+)?#([Ee][-+]?[0-9_]+)? {
                                    ECHO; BEGIN Z; return(numeric_literal);}
[0-9][0-9_]*\:[0-9a-fA-F_]+([.][0-9a-fA-F_]+)?\:([Ee][-+]?[0-9_]+)? {
                                    ECHO; BEGIN Z; return(numeric_literal);}
```

```
\"([^"]*(\"\")*)*\"          {ECHO; BEGIN Z; return(string_literal);}
\%([^%]*(\%\%)*)*\%          {ECHO; BEGIN Z; return(string_literal);}
<Z>\'([^']|\'\')\'           {ECHO; BEGIN Z; return(character_literal);}

[ \t]                        ECHO;          /* ignore whitespace */
"--".*                       ECHO;          /* ignore comments to end-of-line */

[\n]                         {ECHO; ++lines;}          /* linenum;          */


%%
```

F. Data Model

This appendix describes the data model for the kernel of an extensible database. It contains the following elements:

- Method
- Operation
- Property
- Type
- Entity

1. Method

```
object type Method is

        supertypes      := Entity ;

        properties are
                implements      : Set ( Operation )     distributed inverse Operation$baseMethod ;
                isTriggerOf     : Set ( Operation )     distributed inverse Operation$triggers ;
                cals            : _array    ;
                calslots        : _array    ;
        end properties;

private

        properties are
                codeAddr        : Integer   ;
                functionName    : String    optional ;
                                  -- generated method name
                isCreate        : Boolean   optional ;
                isIterator      : Boolean   optional ;
        end properties;

        object MethodCount      : Integer := 0  ;
        object MethodNames      : UnorderedDictionary ;

end Method ;
```

2. Operation

```
object type Operation is

        supertypes      := Type ;
        isInstantiable  := $False ;

        properties are
                baseType        : Type    inverse Type$defops ;
                baseMethod      : Method    inverse Method$implements ;
                triggers        : Set ( Method )     inverse Method$isTriggerOf distributed ;
                isRefineable    : Boolean    ;
                refineeName     : String    ;
                refinesOp       : Operation    inverse Operation$refinements ;
                refinements     : Set ( Operation )     inverse Operation$refinesOp distributed ;
                args            : Set ( ArgSpec )     distributed ;
                returns         : ArgSpec    ;
                exceptions      : Set ( Exception )     distributed ;
        end properties;

        object procedure Effect ( op      : Operation    ;
                args            : ArgumentList    ;
                forwardRef      : Entity    ;
                author          : Entity    ) is
        end Effect ;

        object function Invoke ( op      : Operation    ;
                args    : ArgumentList    ) return Entity  raises ( MissingArgument ,
                InvalidArgument ,
                ExtraneousArgument ,
                InvalidException ,
                UnlinkedMethod ) is
        end Invoke ;

        object procedure setupFreeop ( op      : Operation    ;
                super   : Type    ) is
                -- used to do opsetup on freeops
        end setupFreeop ;
        object function Create ( Ty      : Type    ;
                optional isRefinable    : Boolean    := $True ;
                optional args           : Array ( ArgSpec )     ;
                optional returns        : ArgSpec    ;
                optional exceptions     : Array ( Exception )     ;
                optional baseType       : Type    ;
                optional baseMethod     : Method    ;
                optional triggers       : Array ( Method )     ;
                optional refineeName    : String    ;
                optional refinesOp      : Operation    ;
                optional supertypes     : Array ( Type )     ;
                optional typemodule     : Module    ;
                optional where          : Entity    ;
                optional howNear        : Clustering    ) return Operation
        raises ( RefinedOperationIsUnrefinable ,
                IllegalRefinementReturn ,
```

```
            IllegalRefinementException ,
            IllegalRefinementAdditionalRequiredArgWithNoDefault ,
            IllegalRefinementMissingRequiredArg ,
            IllegalRefinementArgVSpec ,
            DuplicateArgNames ,
            BadDispatchArgSpec ,
            NoMemory ,
            ConflictingRequirements ) is
            type            := CreateOp ;
            supertypes      := Type$Create ;
        end Create ;

        properties are
            dispatchTableIndex      : Integer
                                    -- this is -1 in freeops
            codeAddrList    : CodeAddrList
            proto_cal       : protocal
        end properties;

        methods are
            Create              : $Entity$GenericCreate  ;
            Create   trigger    : Operation_Validate  ;
            Create   trigger    : Operation_Generate  ;
            Effect              : Operation_Effect  ;
            Invoke              : proc_invoke  ;
            setupFreeop         : Operation_setupFreeop  ;
        end methods;

end Operation ;
```

3. Property

```
object type Property is

        supertypes      := Type ;

        properties are
                basetype        : Type   inverse Type$defprops disallows ( set ) ;
                vspec           : Valuespec   ;
                constrainableTo : Valuespec   optional ;
                defaultvalue    : Entity   optional ;
                isOptional      : Boolean   := False disallows ( set ) ;
                isDistributed   : Boolean   := False disallows ( set ) ;
                isRefinable     : Boolean   := True ;
                get             : Operation   optional disallows ( set ) unrefinable ;
                set             : Operation   optional disallows ( set ) unrefinable ;
                init            : Operation   optional disallows ( set ) unrefinable ;
                insert          : Procedure   optional disallows ( set ) unrefinable ;
                remove          : Procedure   optional disallows ( set ) unrefinable ;
                refineeName     : String   optional ;
                refinesProp     : Property   optional inverse Property$refinement ;
                refinements     : Set ( Property )   distributed inverse Property$refinesProp ;
                inverseProp     : Property   optional disallows ( set ) inverse Property$inverseProp ;
        end properties;

end Property ;
```

4. Type

```
object type Type is

    supertypes      := Valuespec ;

    properties are
        name            : symbol    optional ;
        supertypes      : Set ( Type )    inverse Type$subtypes distributed ;
        subtypes        : Set ( Type )    inverse Type$supertypes distributed ;
        defops          : Set ( Operation )    inverse Operation$basetype distributed ;
        defprops        : Set ( Property )    inverse Property$basetype distributed ;
        isInstantiable  : Boolean   := True optional disallows ( set ) ;
        typeclass       : Class   optional allows ( get ) ;
        typevspec       : Valuespec   allows ( get ) ;
        createOp        : $CreateOp   optional inverse $CreateOp$basetype disallows ( set )
        ;
        deleteOp        : Procedure   optional disallows ( set ) ;
        parameters      : Array ( Property )    optional ;
    end properties;

    operations are
        function Instantiate ( t        : Type      ; argl      : ArgumentList     ) return Entity
        raises (  TypeDoesNotExist ,
                  TypeUninstantiable ,
                  TypeHasNoCreateOp ,
                  MissingInitializor ,
                  ExtraneousInitializor ,
                  InvalidArgument ,
                  InvalidException ,
                  UnlinkedCreateOpMethod ,
                  BadCreateOfSystemType ,
                  BadCreateOfApplicationType ) ;
    end operations;

    object function Create ( Ty     : Type   ;
        optional supertypes     : Array ( Type )    ;
        optional name           : Symbol   ;
        optional isInstantiable : Boolean   := True ;
        optional typemodule     : Module   ;
        optional defprops       : Array ( Property )    ;
        optional defops         : Array ( Operation )    ;
        optional RequirementsDesc       : Array ( ReqDesc )    ;
        optional typeclass      : Class   ;
        optional createOp       : CreateOp   ;
        optional deleteOp       : Procedure   ;
        optional parameters     : Array ( Property )    ;
        optional where          : Entity   ;
        optional howNear        : Clustering    ) return Type  raises ( NoMemory ,
        ConflictingRequirements ) is
        type            := CreateOp ;
        supertypes      := Valuespec$create ;
    end Create ;
```

```
end Type ;
```

5. Entity

```
object type Entity is

        directType      := MasterType ;
        supertypes      := Entity ;
        instantiable    := False ;

        properties are
                directType      : Type    disallows ( set ) ;
        end properties;

        operations are
                function equal ( e1     : Entity    ; e2     : Entity   ) return Boolean  ;
                procedure delete ( e      : Entity    ) raises ( CannotDelete ) ;
                function hasType ( e      : Entity    ; typ    : Type   ) return Boolean  ;
                function typeOf ( e     : Entity    ) return Type  ;
                iterator getTypes ( e     : Entity    ) yields Type  ;
        end operations;

        object function GetName ( e      : Entity   ) return Symbol  raises ( NotFound ) is
        end GetName ;

        object function HardReference ( e      : Entity   ) return Entity  raises ( SymbolUndefined ,
                NoValue ) is
        end HardReference ;

        object function Create ( type    : Type   ;
                optional where  : Entity   ;
                optional hownear: Clustering   ) return Entity  raises ( BadCreate ,
                UnInitableProperty ,
                NoMemory ,
                ConflictingRequirements ) is
                type    := CreateOp ;
        end Create ;

end Entity ;
```

END

DATE

FILMED

FEB.

1988